In Terms of Nets

System Design with Petri Nets and Process Algebra

Twan Basten

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Basten, Antonius André

In terms of nets : system design with Petri nets and process algebra /
by Antonius André Basten. - Eindhoven :
Eindhoven University of Technology, 1998. -x, 237 p.
Proefschrift. - ISBN 90-386-0631-1
NUGI 852
Subject headings: Petri nets / process algebra / object oriented modelling /
software description ; program verification
CR Subject Classification (1998): D.2.2, D.2.4, F.3, D.1.3, D.1.5, D.3.3

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA Dissertation Series 1998-05

Printed by University Press Facilities, Eindhoven
Cover Design by Ben Mobach

In Terms of Nets

System Design with Petri Nets and Process Algebra

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op donderdag 3 december 1998 om 16.00 uur

door

Antonius André Basten

geboren te Sevenum

Dit proefschrift is goedgekeurd door de promotoren:
prof.dr. J.C.M. Baeten
prof.dr. K.M. van Hee

" 'What a shame,' Kalten sighed. 'So many good ideas have to be discarded simply because they won't work.' "

The Sapphire Rose, David Eddings

Preface

When I applied for a Ph.D. position almost five years ago, I thought that I knew what it meant to be doing research. I also thought that I understood what it meant to be working towards a Ph.D. thesis. However, research is harder than I expected and successfully completing a Ph.D. thesis is even more difficult than doing research.

As a Ph.D. student, I started working in the Formal Methods group of Jos Baeten and the Information Systems group of Kees van Hee. Jos and Kees wanted to find out whether their favorite formalisms for describing and analyzing concurrent systems, namely process algebra and Petri nets, could be integrated. At the time, several members of both groups were interested in this topic. Thus, I had a very good start in my first year. The results that I obtained during that year form the basis of Chapter 3 of this thesis.

Unfortunately, as any Ph.D. student experiences sooner or later, the good times did not last. Jos became Dean of the Faculty of Mathematics and Computing Science and Kees went to Bakkenist Management Consultants. In addition, the faculty had to cut back costs and was being reorganized. The atmosphere was spoiled and the interest in the topic of my research diminished. At that time, I had serious doubts whether I made the right choice to become a Ph.D. student.

However, times changed, the reorganization was finished, and I found a new interesting topic to work on. In a few months time, I developed the theory which forms the core of Chapter 4.

The next difficult step was to start writing my thesis. I did not like the idea that I had to redo the work of the first three years yet another time. I had already written down all results several times in the form of reports and papers. I felt the work was finished. However, once I had written the first sections of my thesis, I realized that there were still several interesting aspects of my research that were not yet completely clear. My enthusiasm returned and I worked longer and longer hours.

But then something happened that changed my entire life. Due to the many hours I spent behind the keyboard, the muscles and nerves in my arms were damaged, to the point that I could no longer type, write, or lift anything. I had to stop working on my thesis, which ultimately led to a delay of almost nine months. However, with the support of colleagues, family, and friends, I worked myself back into life. The fact that, today, my thesis is finished is one of the signs that I have succeeded.

As may be clear from the above paragraphs, this thesis would not have been the same without the help and support of many people.

First of all, I would like to thank my promotors Jos Baeten and Kees van Hee. From

Jos, I learned many things that are useful when working in a scientific environment. Kees, with his questions and criticism, made sure that I stayed in touch with practice.

Second, I am grateful to Marc Voorhoeve and Wil van der Aalst. My cooperation with Marc and Wil provided the basis for Chapters 3 and 4 of this thesis. I would like to thank Marc for always believing in my research, even when I did not.

I want to thank Eike Best and Gregor Engels for evaluating my thesis as members of the reading committee. I thank Wil van der Aalst, Paul De Bra, Jan Friso Groote, Martin Rem, and Marc Voorhoeve for their willingness to participate in the graduation committee.

Several colleagues and friends commented on various drafts of this thesis or parts of it: Jos Baeten, Eike Best, Gregor Engels, Reiko Heckel, Kees van Hee, Isabelle Reymen, Bram Stappers, Eric Verbeek, and Marc Voorhoeve. I am grateful for their effort. Their suggestions helped to improve the readability and technical presentation of this thesis.

I would like to thank Bram Stappers for writing a script to format the index. Eric Verbeek provided me with the Petri net of Figure 4.3.49 on Page 195 and showed me that the original example was incorrect.

I am indebted to Dragon Systems for developing the speech recognition software that I used to complete this thesis.

The successful completion of a Ph.D. thesis requires a lot of hard work. However, one other aspect is at least as important: I would like to thank my family and close friends for their enthusiasm and interest in good times and their care and support in difficult times.

I thank Bram for having to live with me when I was too busy writing my thesis. I am grateful to Karel and Balkie for the pleasant atmosphere they created at De Koppele.

I want to thank Bram, Eric, Marielle, Eveline, Chiel, Marcel, Gabriëlle, Patricia, and Yves for helping me with all those things that I can no longer do myself.

I am grateful to my parents, Theo and Marga, for always being there when I need them. Their education gave me the attitude and the sense of responsibility that is needed to earn the doctoral degree.

Finally, I would like to say thanks to Isabelle for her love and support. There are so many things she did for me that it is impossible to mention them all. Therefore, I'll keep it simple: Thank you, Isabelle!

Twan Basten

Eindhoven, October 2, 1998

"Life is what happens to you while you're busy making other plans."
John Lennon

Contents

1

Introduction

"*The term "formal methods" denotes software development and analysis activities that entail a degree of mathematical rigor.* (...) *A formal method manipulates a precise mathematical description of a software system for the purpose of establishing that the system does or does not exhibit some property, which is itself precisely defined.*" (DILLON AND SANKAR, 1997)

Motivation. To date, many different formal methods exist to support the development of complex software and hardware systems. Each of these methods builds upon a mathematical formalism to describe and analyze the system under consideration. The motivation for the development of a formal method varies widely. Often a method is developed for a specific application domain. Sometimes methods are based on certain proof techniques. Other formal methods are geared towards the verification of a specific class of properties of systems. Yet other methods are specifically developed to facilitate automated tool support. In a few occasions, mathematical elegance is the main motivation. The result is a wide variety of formal methods and accompanying mathematical formalisms for describing complex systems, each with their own strengths and weaknesses.

Despite the large number of formal methods existing to date, new methods are still being developed on a regular basis. Often, the question arises whether it is worth the effort. There is still much that can be improved on and learned from existing methods and formalisms for developing complex systems. It is particularly fruitful to study *combinations* of several methods or formalisms with different characteristics and complementary strengths. Such studies may lead to improved and more complete methods and formalisms, as well as a better conceptual understanding of the methods and formalisms, their strengths, and their weaknesses.

This thesis studies the combination of two formalisms for reasoning about complex systems, namely Petri nets and process algebra. Petri nets originate from (PETRI, 1962), whereas the process algebra considered in this thesis was first described in (BERGSTRA

AND KLOP, 1982). Petri nets and process algebra share two important characteristics. First, they both have a precise mathematical definition. Second, they are both designed for reasoning about so-called *concurrent* systems. A concurrent system is a system that exhibits parallelism or distribution, either physically or in its behavior. The most important characteristic of a concurrent system is that it usually consists of multiple components that communicate to exchange information or to synchronize. Very many of the complex systems being developed in current software-engineering practice belong to the class of concurrent systems.

Except for the abovementioned two common characteristics, Petri nets and process algebra are entirely different formalisms. Petri nets have an intuitive graphical representation, which is easy to understand for non-experts. A Petri-net model describes both the states and the actions of the system under consideration. There are many analysis techniques available to investigate properties about the states as well as the dynamic behavior of a Petri-net model. Process algebra is a purely symbolic formalism which is particularly well suited for computer manipulation. A process-algebraic description of a system focuses on its dynamic behavior. It usually has no explicit representation of system states. Proof techniques in process algebra are generally aimed at showing the equality of behavioral descriptions. Consequently, process algebra is useful to compare behavioral descriptions of concurrent systems. For example, it is often used to compare a high-level specification of a system and a more detailed description of an implementation of the system. The goal of such a comparison is to verify whether the implementation satisfies the specification.

The above observations show that, in several aspects, Petri nets and process algebra are complementary. The goal of this thesis is to study topics combining the strengths of both Petri nets and process algebra. An important objective is to improve the acceptance of formal methods in design practice.

Considering current design practice, the most important problem is to cope with the increasing complexity of the systems being developed. Therefore, a lot of effort is put into the development of methods that support modular or component-based design. Important goals of such methods are to structure the design process and to stimulate the reuse of system components during the design process and in future designs. As explained below, the two main topics of this thesis are both inspired by these developments in design practice.

To conclude this paragraph, it is useful to explain what is *not* the aim of this thesis. It is not meant to argue the need for formal methods in the design of complex software and hardware systems. It is also beyond the scope of this thesis to give a detailed overview and comparison of formal methods in general. The interested reader is referred to (BOWEN, 1998), which contains numerous references to literature on these two topics.

Overview. The remainder of this thesis is organized as follows. Chapter 2 contains an introduction to basic concurrency theory. It gives a detailed and fairly complete introduction to Petri nets and process algebra, including references to related literature. The chapter is aimed at readers with a basic mathematical knowledge. It is set up in such a way that the thesis is self-contained. Prior knowledge of Petri nets or process algebra is not necessary.

Chapter 3 describes the first contribution of this thesis. It combines Petri nets and process algebra into a method supporting compositional design. In this method, Petri nets are used to model system components. Process algebra is used to specify and verify the behavior of these components. The method is compositional in the sense that the behavior of the entire system can be derived from the behavior of its components. The basis for the method is formed by a number of algebraic semantics for the behavior of so-called modular P/T nets. The class of modular P/T nets is a class of Petri nets especially developed to support modular design. The chapter describes two case studies to evaluate the method in practical design situations. The chapter is based on results published earlier in (BASTEN AND VOORHOEVE, 1995a, 1995b).

Chapter 4 is devoted to the second topic of this thesis. It describes an approach to support inheritance of dynamic behavior in object-oriented design methods. In such methods, the so-called class construct is used to structure a system design. A class describes a set of objects sharing a common static structure and dynamic behavior. The concept of inheritance is an important means to stimulate the reuse of class descriptions during the design process. However, many object-oriented methods being used in practice lack a precise definition of inheritance of dynamic behavior. In Chapter 4, inheritance of dynamic behavior is first studied in a simple process-algebraic setting. The goal is to get a clear understanding of the important concepts. The results are then translated to a framework based on Petri nets. The Petri-net framework is more expressive than the algebraic framework and it is closer to existing object-oriented methods. The chapter also contains a small case study to validate the approach taken to support inheritance of dynamic behavior. The chapter is based on results published earlier in (BASTEN AND VAN DER AALST, 1996; VAN DER AALST AND BASTEN, 1997).

Chapters 3 and 4 both contain detailed introductions, including a further motivation of the research, and detailed conclusions, including discussions of related work and open problems. Chapter 5 contains some concluding remarks discussing the results of this thesis in the context of system design as sketched in this introductory chapter.

Finally, this thesis ends with an overview of algebraic axioms, which is a useful aid in reading algebraic verifications, the bibliography, the index, a summary of the thesis, a translation of the summary into Dutch, and a brief curriculum vitae of the author.

2

Basic Concurrency Theory

2.1 Introduction

Two of the most widely used formalisms for describing concurrent systems are Petri nets and process algebra. This chapter provides an introduction to concurrency theory, containing basic definitions and results. A concurrent system is assumed to be a collection of processes running in parallel that may communicate with each other to synchronize or to exchange information. Three different formalisms for describing concurrent systems are discussed. First, a general framework of processes is introduced, based on the notion of transition systems. Transition systems can be used to provide different formalisms for describing concurrent systems with a common operational semantics. In this way, it is possible to compare systems described in any such formalism in an unambiguous way. Second, Section 2.3 introduces the formalism of Place/Transition nets, or simply P/T nets. The framework of P/T nets is a basic class of Petri nets which is illustrative for other Petri-net formalisms. Third, in Section 2.4, a few simple process algebras are introduced to illustrate the essentials of a process-algebraic approach to describing concurrent systems. Finally, Section 2.5 contains some concluding remarks.

2.2 Process Theory

2.2.1 Operational semantics

A very natural and straightforward way to describe a concurrent system is by means of a *labeled transition system*. A labeled transition system is a set of *states* plus a *transition relation* on states. Each transition is labeled with an *action*. An action may be any kind of activity performed by a concurrent system, such as a computation local to some component of the system, a procedure call, the sending of a message or the receipt of a message. For modeling purposes, the details of actions are often not important. Actions are assumed to be atomic entities without internal structure. The set of states in a labeled transition system is an abstraction of all possible states of a concurrent system. The transition relation describes the change in the state of a concurrent system when some action, the label of the transition, is performed.

The basic notion in the framework of labeled transition systems used in this thesis is the so-called *process space*. A process space describes a *set of processes*. A process space is a labeled transition system as described above extended with a termination predicate on states. Each state in a process space can be interpreted as the initial state of a process. A *process* is a labeled transition system extended with a termination predicate and a distinguished initial state. The termination predicate of a process defines in what states the process *can terminate successfully*. If a process is in a state where it cannot perform any actions or terminate successfully, then it is said to be in a *deadlock*. The possibility to distinguish between successful termination and deadlock is sometimes useful. Process spaces form the basis for any operational semantics for concurrent systems. A predicate on the elements of some set is represented as a subset of this set: It holds for elements in the subset and it does not hold for elements outside the subset.

Definition 2.2.1. (Process space) A *process space* is a quadruple $(\mathcal{S}, \mathcal{A}, \longrightarrow, \downarrow)$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\_ \xrightarrow{\_} \_ \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is a ternary transition relation, and $\downarrow \_ \subseteq \mathcal{S}$ is a termination predicate.

Let $(\mathcal{S}, \mathcal{A}, \longrightarrow, \downarrow)$ be some process space. Each state s in $\mathcal{S}$ uniquely determines a process that consists of all states reachable from s.

Definition 2.2.2. (Reachability) The *reachability relation* $\_ \stackrel{*}{\Longrightarrow} \_ \subseteq \mathcal{S} \times \mathcal{S}$ is defined as the smallest relation satisfying, for any $s, s', s'' \in \mathcal{S}$ and $\alpha \in \mathcal{A}$,

$s \stackrel{*}{\Longrightarrow} s$ and

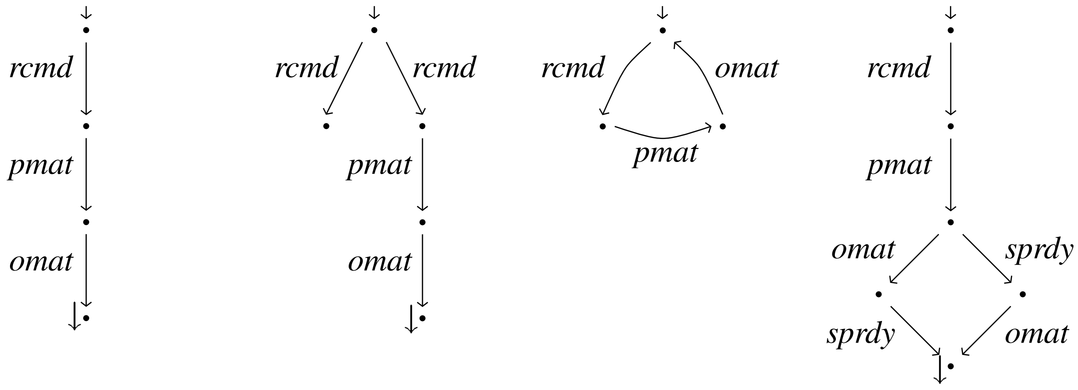
$(s \stackrel{*}{\Longrightarrow} s' \wedge s' \xrightarrow{\alpha} s'') \;\Rightarrow\; s \stackrel{*}{\Longrightarrow} s''$.

State s' is said to be *reachable* from state s if and only if $s \stackrel{*}{\Longrightarrow} s'$. The set of all states reachable from s is denoted $s*$.

Definition 2.2.3. (Process) Let s be a state in $\mathcal{S}$. The *process* defined by s is the 5-tuple $(s, s*, \mathcal{A}, \longrightarrow \cap (s* \times \mathcal{A} \times s*), \downarrow \cap s*)$. State s is the *initial state* of the process.

In the remainder, processes are identified with their initial states. Sometimes, it is intuitive to think about elements of $\mathcal{S}$ as states, whereas sometimes it is more natural to see them as processes.

Example 2.2.5. Figure 2.2.4 on the facing page shows some examples of processes modeling variants of a very simple production unit. Process states are depicted as dots. The initial state of a process is marked with a small incoming arrow. The production unit in Figure 2.2.4(a) is a simple sequential process that starts with receiving a command by performing action *rcmd*. After it has received a command, it processes some input material (*pmat*), outputs the processed material (*omat*), and then terminates. The unit in Figure 2.2.4(b) is the same process, except that, upon receiving a command, it may deadlock.



(a) (b) (c) (d)

Figure 2.2.4: Some simple examples of processes.

A reason could be that the command is not understood. The process in Figure 2.2.4(c) is a variant that iterates the behavior of the production unit in (a). It does not have an option to terminate. The production unit of Figure 2.2.4(d) is again a non-iterative process. The interesting aspect of this process is that it exhibits concurrency. In parallel to its output action, it sends a ready-processing signal (*sprdy*) to, for example, an operator. This can be useful for an operator who must pick up the processed material if processing takes a large, but variable amount of time. In the context of process spaces, concurrency typically means that actions may be executed in any order. This representation of concurrent behavior is often called the *interleaving* view of concurrency.

As mentioned, process spaces are used to provide a formalism for describing concurrent systems with an operational semantics. However, it might occur that the behavior of two different processes is very similar, even to such an extent that the processes may be considered equivalent. Therefore, a process space is equipped with an equivalence relation on processes, yielding the following definition for an operational semantics. The definition provides sufficient flexibility to precisely define the meaning of system specifications in any Petri-net formalism or any process algebra used in this thesis.

Definition 2.2.6. (Operational semantics) An *operational semantics* for some given formalism for describing concurrent systems is a process space with an accompanying equivalence relation on processes.

In the next two subsections, two well-known equivalence relations on processes are discussed that capture the notion of similarity of behavior in an intuitive way. The basic idea is that two processes are equivalent if and only if they can always copy, or *simulate*, each others actions. Two processes satisfying this requirement are called *bisimilar*. Bisimilarity was first introduced in (PARK, 1981). Although there are many possible choices for a meaningful equivalence relation on processes, bisimilarity is the basic semantic equivalence used throughout this thesis. There are several reasons for choosing bisimilarity.

First, it has an easy-to-understand, intuitive definition. Second, it preserves some important properties of concurrent systems, such as deadlocks. And finally, processes that are bisimilar are also equivalent in many other semantics for concurrent systems. However, it is only fair to note that bisimilarity is not always the proper equivalence relation for concurrent systems.

> "*When semantic equivalences are used in the design of concurrent systems* (…), *they should be chosen in such a way that two system descriptions are considered equivalent only if the described behaviours share the properties that are essential in the context in which the system will be embedded.*"
>
> (VAN GLABBEEK, 1990a, Introduction)

For a detailed study of many different semantics for concurrent systems, the reader is referred to (VAN GLABBEEK, 1990a, 1990b, 1993). Since bisimilar processes are also equivalent in many other semantics, the ideas presented in this thesis carry over to these semantics with only little change.

In the next subsection, the formal definition of bisimilarity is given and it is shown that it is an equivalence relation. Bisimilarity is often referred to as *strong* bisimilarity in order to distinguish it from other forms of bisimilarity. In Section 2.2.3, the notion of *branching* bisimilarity is introduced. Branching bisimilarity is a variant of bisimilarity that distinguishes *external* or *observable* behavior from *internal* or *silent* behavior. The distinction between external and internal behavior captures the idea that an environment observing two processes might not be able to see any differences in their behavior while internally the two processes perform different computations.

2.2.2 Strong bisimilarity

Let $(\mathcal{P}, \mathcal{A}, \longrightarrow, \downarrow)$ be some process space as defined in Definition 2.2.1 on Page 6. The notation has been changed from $\mathcal{S}$ to $\mathcal{P}$ to emphasize that, in this subsection, it is most natural to see elements of $\mathcal{P}$ as *processes*.

Definition 2.2.7. (Strong bisimilarity) A binary relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is called a *bisimulation* or *strong bisimulation* if and only if, for any $p, p', q, q' \in \mathcal{P}$ and $\alpha \in \mathcal{A}$,

i) $p\mathcal{R}q \wedge p \xrightarrow{\alpha} p' \Rightarrow (\exists q' : q' \in \mathcal{P} : q \xrightarrow{\alpha} q' \wedge p'\mathcal{R}q')$,

ii) $p\mathcal{R}q \wedge q \xrightarrow{\alpha} q' \Rightarrow (\exists p' : p' \in \mathcal{P} : p \xrightarrow{\alpha} p' \wedge p'\mathcal{R}q')$, and

iii) $p\mathcal{R}q \Rightarrow (\downarrow p \Leftrightarrow \downarrow q)$.

Two processes p and q are called *bisimilar*, or *strongly bisimilar*, denoted $p \sim q$, if and only if there exists a bisimulation $\mathcal{R}$ such that $p\mathcal{R}q$.

Figure 2.2.8 on the facing page shows the essence of a bisimulation between two processes. It clearly shows that bisimulation captures the intuitive idea that two processes are equivalent if and only if, at any given point in time, they can copy each others actions. The third requirement in the definition above states that two bisimilar processes must have the same termination behavior.

$$\begin{tikzcd}
p \arrow[r, dash, dashed, "\mathcal{R}"] \arrow[d, "\alpha"'] & q \arrow[d, "\alpha"] \\
p' \arrow[r, dash, dashed, "\mathcal{R}"'] & q'
\end{tikzcd}$$

Figure 2.2.8: The essence of a bisimulation.

Theorem 2.2.9. *Strong bisimilarity,* $\sim$ *, is an equivalence relation.*

Proof. It must be shown that bisimilarity is reflexive, symmetric, and transitive.

Reflexivity: Let $\mathcal{I}$ be the identity relation on $\mathcal{P} \times \mathcal{P}$. Obviously, $\mathcal{I}$ is a bisimulation. Hence, for any $p \in \mathcal{P}$, $p \sim p$.

Symmetry: Let p and q be processes in $\mathcal{P}$. Let $\mathcal{R}$ be a bisimulation such that $p\mathcal{R}q$. It follows from the symmetry in Definition 2.2.7 on the preceding page that $\mathcal{R}^{-1}$, the inverse of $\mathcal{R}$, is a bisimulation such that $q\mathcal{R}^{-1}p$. Hence, for any $p, q \in \mathcal{P}$, $p \sim q$ implies $q \sim p$.

Transitivity: Let p, q, and r be processes in $\mathcal{P}$. Let $\mathcal{Q}$ and $\mathcal{R}$ be bisimulations such that $p\mathcal{Q}q$ and $q\mathcal{R}r$. It is straightforward to verify that the relation composition $\mathcal{Q} \circ \mathcal{R}$ is a bisimulation such that $p\mathcal{Q} \circ \mathcal{R}r$. Hence, for any $p, q, r \in \mathcal{P}$, $p \sim q$ and $q \sim r$ implies $p \sim r$.

$\square$

Example 2.2.10. Consider again the examples of Figure 2.2.4(a) and (b) on Page 7. It is not difficult to verify that it is impossible to construct a bisimulation between these two processes. This observation is consistent with the claim made earlier that bisimilarity preserves deadlocks in processes. That is, two processes can only be bisimilar if they have the same deadlocks (see, for example, BAETEN AND WEIJLAND, 1990, Chapter 7).

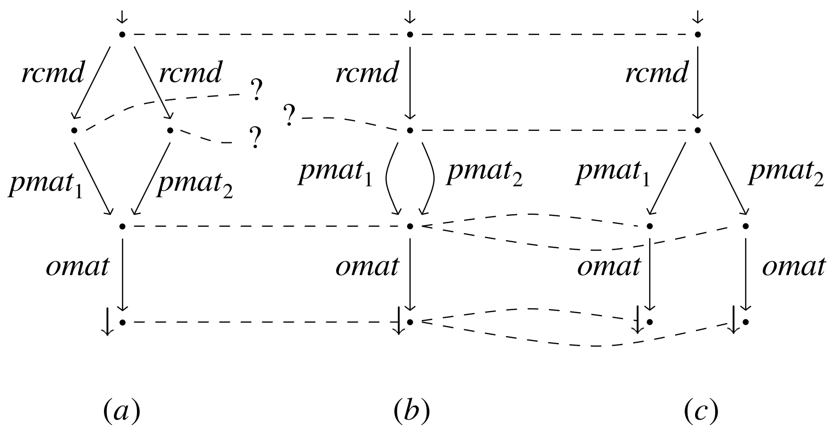


Figure 2.2.11: Bisimilar or not?

Example 2.2.12. Figure 2.2.11 on the previous page shows again three examples of processes modeling a production unit. Although all three are different, they appear to behave similarly. In these examples, a unit has two different material sources. Hence, it has two different processing commands $pmat_1$ and $pmat_2$. In the version of the unit depicted in Figure 2.2.11(a), the command specifies which material source should be used. Therefore, before the unit receives its command, the choice is made between the two sources. In the unit depicted in Figure 2.2.11(b), the command does not specify the source and the choice is made after the unit has received its command. It is not difficult to see that it is impossible to construct a bisimulation between the two processes. The reason is that the *moments of choice* of the two processes are different. In other words, bisimilarity is said to preserve the *branching structure* of processes (see, for example, VAN GLABBEEK, 1994). It is useful to distinguish the two processes in Figure 2.2.11(a) and (b). Consider, for example, a situation where only the first input source is filled with material. If the unit of Figure 2.2.11(a) receives a command specifying the second source, it will not be able to execute its processing task. If at least one of the two input sources contains material, the unit in Figure 2.2.11(b) is always able to process.

Figure 2.2.11 shows also a third version of the production unit. It is easy to see that the two processes in Figure 2.2.11(b) and (c) are bisimilar. The dashed lines depict what pairs of processes construct a bisimulation. It is straightforward to check that it satisfies the conditions of Definition 2.2.7 on Page 8. The two processes have the same branching structure. The only difference is that in process (c) the two branches do not join, whereas they do come together in the process depicted in (b).

2.2.3 Branching bisimilarity

To be able to distinguish between external, or observable, and internal, or silent, behavior, *silent actions* are introduced. Silent actions are actions that cannot be observed. Usually, silent actions are denoted with the action label τ. A single symbol is sufficient, since all internal actions are equal in the sense that they do not have any visible effects. Since strong bisimilarity as defined in the previous subsection does not distinguish between observable actions and silent actions, it is not a suitable equivalence for processes with silent behavior. In case one is interested in observable behavior only, processes with the same observable behavior, but with different internal behavior should be equal. Branching bisimilarity is an equivalence that satisfies this requirement. Branching bisimilarity was first introduced in (VAN GLABBEEK AND WEIJLAND, 1989). The definition given in this subsection is slightly different from the original definition. In fact, it is the definition of *semi*-branching bisimilarity, which was first defined in (WEIJLAND, 1989, Chapter 1). It can be shown that the two notions are equivalent in the sense that they define the same equivalence relation on processes (VAN GLABBEEK AND WEIJLAND, 1996; BASTEN, 1996). The reason for using the alternative definition is that it is more concise and more intuitive than the original definition. It also yields shorter proofs. A comparison of the two definitions can be found in (BASTEN, 1996).

As is the case for strong bisimilarity, branching bisimilarity distinguishes processes with different moments of choice (see VAN GLABBEEK, 1994). It is a slightly finer equivalence than the well-known observation equivalence (MILNER, 1980, 1989). That is, it distinguishes more processes than observation equivalence (see VAN GLABBEEK, 1993). A comparison of branching bisimilarity, observation equivalence, and a few other equivalences on processes with silent behavior can be found in (VAN GLABBEEK AND WEIJLAND, 1996).

Let $(\mathcal{P}, \mathcal{A}, \longrightarrow, \downarrow)$ be a process space as defined in Definition 2.2.1 on Page 6. The set of actions $\mathcal{A}$ is defined as $A \cup \{\tau\}$, where A is some set of observable actions. To define branching bisimilarity, two auxiliary definitions are needed: *i*) a relation expressing that a process can evolve into another process by executing a sequence of zero or more τ actions; *ii*) a predicate expressing that a process can terminate by performing zero or more τ actions.

Definition 2.2.13. The relation $\_ \Longrightarrow \_ \subseteq \mathcal{P} \times \mathcal{P}$ is defined as the smallest relation satisfying, for any $p, p', p'' \in \mathcal{P}$,

$p \Longrightarrow p$ and

$(p \Longrightarrow p' \wedge p' \xrightarrow{\tau} p'') \Rightarrow p \Longrightarrow p''$.

Definition 2.2.14. The predicate $\Downarrow \_ \subseteq \mathcal{P}$ is defined as the smallest set of processes satisfying, for any $p, p' \in \mathcal{P}$,

$\downarrow p \Rightarrow \Downarrow p$ and

$(\Downarrow p \wedge p' \xrightarrow{\tau} p) \Rightarrow \Downarrow p'$.

Note that it is also possible to define the predicate $\Downarrow$ in terms of $\Longrightarrow$.

Let, for any processes $p, p' \in \mathcal{P}$ and action $\alpha \in \mathcal{A}$, $p \xrightarrow{(\alpha)} p'$ be an abbreviation of the predicate $(\alpha = \tau \wedge p = p') \vee p \xrightarrow{\alpha} p'$. Thus, $p \xrightarrow{(\tau)} p'$ means that zero τ actions are performed, when the first disjunct of the predicate is satisfied, or that one τ action is performed, when the second disjunct is satisfied. For any observable action $a \in A$, the first disjunct of the predicate can never be satisfied. Hence, $p \xrightarrow{(a)} p'$ is simply equal to $p \xrightarrow{a} p'$, meaning that a single a action is performed.

Definition 2.2.15. ((Rooted) branching bisimilarity) A binary relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ is called a *branching bisimulation* if and only if, for any $p, p', q, q' \in \mathcal{P}$ and $\alpha \in \mathcal{A}$,

i) $p\mathcal{R}q \wedge p \xrightarrow{\alpha} p' \Rightarrow$
$(\exists\, q', q'' : q', q'' \in \mathcal{P} : q \Longrightarrow q'' \xrightarrow{(\alpha)} q' \wedge p\mathcal{R}q'' \wedge p'\mathcal{R}q')$,

ii) $p\mathcal{R}q \wedge q \xrightarrow{\alpha} q' \Rightarrow$
$(\exists\, p', p'' : p', p'' \in \mathcal{P} : p \Longrightarrow p'' \xrightarrow{(\alpha)} p' \wedge p''\mathcal{R}q \wedge p'\mathcal{R}q')$, and

iii) $p\mathcal{R}q \Rightarrow (\downarrow p \Rightarrow \Downarrow q \wedge \downarrow q \Rightarrow \Downarrow p)$.

Two processes are called *branching bisimilar*, denoted $p \sim_b q$, if and only if there exists a branching bisimulation $\mathcal{R}$ such that $p\mathcal{R}q$.

A branching bisimulation $\mathcal{R}$ is called a *rooted* branching bisimulation between p and q in $\mathcal{P}$ if and only if $p\mathcal{R}q$ and, for any $p', q' \in \mathcal{P}$ and $\alpha \in \mathcal{A}$, *iv*) $p \xrightarrow{\alpha} p' \Rightarrow (\exists\, q' : q' \in \mathcal{P} : q \xrightarrow{\alpha} q' \wedge p'\mathcal{R}q')$,

v) $q \xrightarrow{\alpha} q' \Rightarrow (\exists\, p' : p' \in \mathcal{P} : p \xrightarrow{\alpha} p' \wedge p'\mathcal{R}q')$, and

vi) $\downarrow p \Leftrightarrow\ \downarrow q$.

Two processes p and q are called *rooted branching bisimilar*, denoted $p \sim_{rb} q$, if and only if there exists a rooted branching bisimulation between p and q.

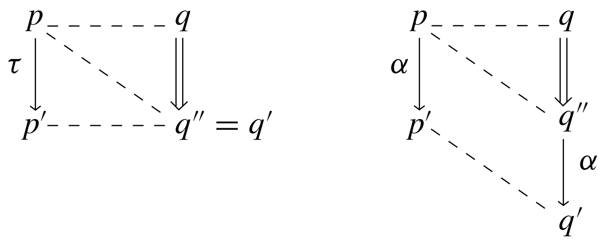


Figure 2.2.16: The essence of a branching bisimulation.

Figure 2.2.16 shows the essence of a branching bisimulation. A process must be able to simulate any action of an equivalent process after performing any number of silent actions, except for a silent action which it may or may not simulate. The third property in Definition 2.2.15 on the previous page guarantees that related processes always have the same termination options. The *root condition*, introduced in the last three requirements of the definition, states that the *initial* actions of two rooted branching bisimilar processes must be the same. The requirements are identical to the requirements in the definition of strong bisimilarity (Definition 2.2.7 on Page 8). The root condition is needed, because branching bisimilarity is not a congruence for the process-algebraic choice operator which is introduced in Section 2.4. Rooted branching bisimilarity, on the other hand, is a congruence for all the process-algebra operators used in this thesis. As explained in Section 2.4, any equivalence used in an operational semantics for a process algebra must have the congruence property for all the algebraic operators. Below, it is shown that branching bisimilarity and rooted branching bisimilarity are equivalence relations. However, before giving these results, a few examples are given to illustrate both equivalences.

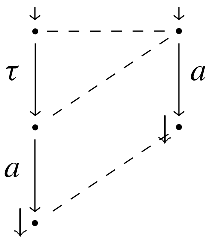


Figure 2.2.17: Two branching bisimilar processes that are not rooted branching bisimilar.

Example 2.2.18. Figure 2.2.17 shows two processes that are branching bisimilar but not *rooted* branching bisimilar. The problem is caused by the initial silent action of the left process, which cannot be simulated by the right one. This example shows the essential difference between branching bisimilarity and rooted branching bisimilarity. Branching bisimilarity allows to remove initial silent actions, whereas rooted branching bisimilarity does not.

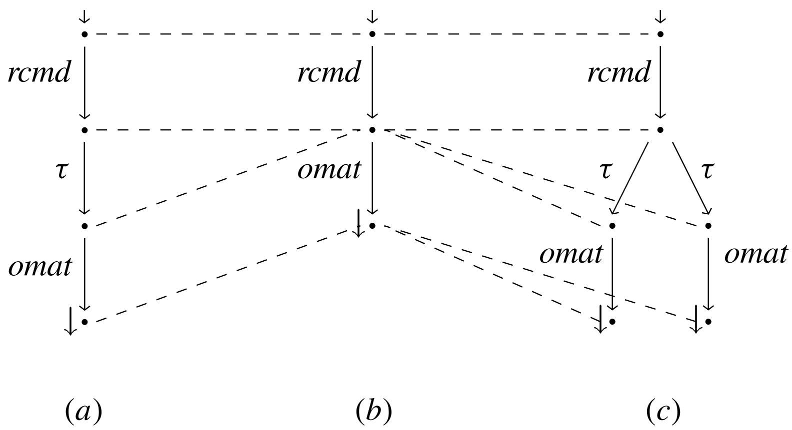


Figure 2.2.19: Some examples of branching bisimilar processes.

Example 2.2.20. Figure 2.2.19 shows three examples of branching bisimilar processes. The processes in (a) and (c) are the processes of Figure 2.2.11 (b) and (c) on Page 9 after abstracting away the processing actions $pmat_1$ and $pmat_2$. Note that the two transitions labeled $pmat_1$ and $pmat_2$ in Figure 2.2.11(b) result in a single transition labeled τ in Figure 2.2.19(a), because it is not possible to have two transitions with the same label between the same process states. The reason for the abstraction could be that one is interested in the input/output behavior of the production units. In both cases, the i/o behavior should be the receipt of a command followed by the output of processed material. Figure 2.2.19 shows that this is indeed the case. It is not difficult to verify that the two relations depicted by the dashed lines are indeed branching bisimulations. Note that they also satisfy the root condition. Hence, the three processes are not only branching bisimilar, but also rooted branching bisimilar.

To prove that branching bisimilarity and rooted branching bisimilarity are equivalence relations, three auxiliary lemmas are needed.

Lemma 2.2.21. *Let p and q be two processes in $\mathcal{P}$; let $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ be a branching bisimulation between p and q. For any $p', q' \in \mathcal{P}$,*

i) $p \Longrightarrow p' \Rightarrow (\exists\, q' : q' \in \mathcal{P} : q \Longrightarrow q' \wedge p'\mathcal{R}q')$ *and*

ii) $q \Longrightarrow q' \Rightarrow (\exists\, p' : p' \in \mathcal{P} : p \Longrightarrow p' \wedge p'\mathcal{R}q')$.

Proof. Since the proofs of implications *i*) and *ii*) are symmetric only implication *i*) is proven. The proof is by induction on the number of silent actions from p to p'.

Base case: Assume the number of silent actions from p to p' equals zero. Then, p and p' must be equal. Since $p\mathcal{R}q$ and $q \Longrightarrow q$, Lemma 2.2.21*i*) is satisfied.

Inductive step: Assume $p \Longrightarrow p'$ in n silent actions, for some $n \geq 1$. It follows from Definition 2.2.13 ($\Longrightarrow$) on Page 11 that there must exist a $p'' \in \mathcal{P}$ such that $p \Longrightarrow p''$ in $n-1$ silent actions and $p'' \xrightarrow{\tau} p'$. By induction, there exists a $q'' \in \mathcal{P}$ such that $q \Longrightarrow q''$ and $p''\mathcal{R}q''$. Since $p'' \xrightarrow{\tau} p'$ and $\mathcal{R}$ is a branching bisimulation, there exist q' and q''' such that $q'' \Longrightarrow q''' \xrightarrow{(\tau)} q'$, $p''\mathcal{R}q'''$, and $p'\mathcal{R}q'$. Hence, $q \Longrightarrow q'$ and $p'\mathcal{R}q'$, which proves Lemma 2.2.21*i*) also in this case. □

Lemma 2.2.22. *Let $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ be a branching bisimulation. For any $p, q \in \mathcal{P}$ such that $p\mathcal{R}q$, $\Downarrow p \Leftrightarrow \Downarrow q$.*

Proof. Only one implication is proven. The other implication follows from a symmetrical argument. The proof is by induction on the number of silent actions that a process needs to execute in order to terminate successfully.

Base case: Let $p, q \in \mathcal{P}$ be two processes such that $p\mathcal{R}q$ and $\Downarrow p$ with zero silent actions. It follows from Definition 2.2.14 ($\Downarrow$) on Page 11 that $\downarrow p$. Hence, Requirement *iii*) of Definition 2.2.15 (Branching bisimilarity) also on Page 11 yields that $\Downarrow q$.

Inductive step: Let $p, q \in \mathcal{P}$ be two processes such that $p\mathcal{R}q$ and $\Downarrow p$ with n silent actions, for some $n \geq 1$. It follows from Definition 2.2.14 ($\Downarrow$) that there must exist a $p' \in \mathcal{P}$ such that $p \xrightarrow{\tau} p'$ and $\Downarrow p'$ with $n-1$ silent actions. Since $\mathcal{R}$ is a branching bisimulation, there must exist $q', q'' \in \mathcal{P}$ such that $q \Longrightarrow q'' \xrightarrow{(\tau)} q'$, $p\mathcal{R}q''$, and $p'\mathcal{R}q'$. Since $\Downarrow p'$ with $n-1$ silent actions, by induction, $\Downarrow q'$. It follows immediately from Definitions 2.2.13 ($\Longrightarrow$) and 2.2.14 ($\Downarrow$), that $\Downarrow q$, which completes the proof. □

As mentioned, the definition of branching bisimilarity chosen in this thesis is slightly different from the original definition. The following lemma is only valid for (rooted) branching bisimulations as defined in Definition 2.2.15 on Page 11. In (BASTEN, 1996), an example is given showing that it does not hold for the original definition in (VAN GLABBEEK AND WEIJLAND, 1989).

Lemma 2.2.23. *The relation composition of two (rooted) branching bisimulations is a (rooted) branching bisimulation.*

Proof. Let p, q, and r be processes in $\mathcal{P}$; let $\mathcal{Q}$ and $\mathcal{R}$ be (rooted) branching bisimulations such that $p\mathcal{Q}q$ and $q\mathcal{R}r$. The goal is to show that $\mathcal{Q} \circ \mathcal{R}$ is a (rooted) branching bisimulation between p and r. First, to prove Requirement *i*) of Definition 2.2.15 ((Rooted) branching bisimilarity) on Page 11, assume there is a $p' \in \mathcal{P}$ such that $p \xrightarrow{\alpha} p'$, for some $\alpha \in \mathcal{A}$. It must be shown that

$$(\exists r', r'' : r', r'' \in \mathcal{P} : r \Longrightarrow r'' \xrightarrow{(\alpha)} r' \wedge p\mathcal{Q} \circ \mathcal{R}r'' \wedge p'\mathcal{Q} \circ \mathcal{R}r'). \tag{2.2.24}$$

Since $p\mathcal{Q}q$, there exist $q', q'' \in \mathcal{P}$ such that $q \Longrightarrow q'' \xrightarrow{(\alpha)} q'$, $p\mathcal{Q}q''$, and $p'\mathcal{Q}q'$. Since $q\mathcal{R}r$ and $q \Longrightarrow q''$, Lemma 2.2.21 *i*) on the previous page yields that there is an $r'' \in \mathcal{P}$ such that $r \Longrightarrow r''$ and $q''\mathcal{R}r''$. Two cases can be distinguished.

i) Assume $\alpha = \tau$ and $q'' = q'$. It follows immediately that $r \Longrightarrow r'' \xrightarrow{(\alpha)} r''$, $p\mathcal{Q} \circ \mathcal{R}r''$, and $p'\mathcal{Q} \circ \mathcal{R}r''$. Hence, Property 2.2.24 is satisfied.

ii) Assume $q'' \xrightarrow{\alpha} q'$. Since $q''\mathcal{R}r''$ and $\mathcal{R}$ is a branching bisimulation, there must exist $r', r''' \in \mathcal{P}$ such that $r'' \Longrightarrow r''' \xrightarrow{(\alpha)} r'$, $q''\mathcal{R}r'''$, and $q'\mathcal{R}r'$. It follows from $r \Longrightarrow r''' \xrightarrow{(\alpha)} r'$, $p\mathcal{Q} \circ \mathcal{R}r'''$, and $p'\mathcal{Q} \circ \mathcal{R}r'$, that also in this case Property 2.2.24 is satisfied.

Second, for reasons of symmetry, it follows immediately from the above argument that for any $r' \in \mathcal{P}$ and $\alpha \in \mathcal{A}$ such that $r \xrightarrow{\alpha} r'$, Requirement *ii*) of Definition 2.2.15 on Page 11 is satisfied.

Third, to prove Requirement *iii*) of Definition 2.2.15, assume that $\downarrow p$. Since $\mathcal{Q}$ is a branching bisimulation such that $p\mathcal{Q}q$, it follows that $\Downarrow q$. Since $\mathcal{R}$ is a branching bisimulation relating q to r, it follows from Lemma 2.2.22 on the preceding page that $\Downarrow r$. Hence, $\downarrow p$ implies $\Downarrow r$. A symmetrical argument yields that Requirement *iii*) of Definition 2.2.15 is satisfied.

Finally, in case $\mathcal{Q}$ and $\mathcal{R}$ are *rooted* branching bisimulations, it is straightforward to verify that also $\mathcal{Q} \circ \mathcal{R}$ satisfies the root condition stated in Requirements *iv*) through *vi*) of Definition 2.2.15. □

Theorem 2.2.25. *Branching bisimilarity, $\sim_b$, and rooted branching bisimilarity, $\sim_{rb}$, are equivalence relations.*

Proof. It must be shown that branching bisimilarity and rooted branching bisimilarity are reflexive, symmetric, and transitive.

Reflexivity: Let $\mathcal{I}$ be the identity relation on $\mathcal{P} \times \mathcal{P}$. It is clear that for any $p \in \mathcal{P}$, $\mathcal{I}$ is both a branching bisimulation and a rooted branching bisimulation between p and p. Hence, for any $p \in \mathcal{P}$, $p \sim_b p$ and $p \sim_{rb} p$.

Symmetry: Let p and q be processes in $\mathcal{P}$. Let $\mathcal{R}$ be a (rooted) branching bisimulation such that $p\mathcal{R}q$. It follows from the symmetry in Definition 2.2.15 on Page 11 that $\mathcal{R}^{-1}$ is a (rooted) branching bisimulation such that $q\mathcal{R}^{-1}p$. Hence, for any $p, q \in \mathcal{P}$, $p \sim_b q$ implies $q \sim_b p$ and $p \sim_{rb} q$ implies $q \sim_{rb} p$.

Transitivity: Let p, q, and r be processes in $\mathcal{P}$. Let $\mathcal{Q}$ and $\mathcal{R}$ be (rooted) branching bisimulations such that $p\mathcal{Q}q$ and $q\mathcal{R}r$. It follows from Lemma 2.2.23 on the preceding page that the relation composition $\mathcal{Q} \circ \mathcal{R}$ is a (rooted) branching bisimulation such that $p\mathcal{Q} \circ \mathcal{R}r$. Hence, for any $p, q, r \in \mathcal{P}$, $p \sim_b q$ and $q \sim_b r$ implies $p \sim_b r$, and $p \sim_{rb} q$ and $q \sim_{rb} r$ implies $p \sim_{rb} r$.

□

2.3 Petri Nets

This section presents an introduction to Petri nets as a formalism for modeling and analyzing concurrent systems. Petri nets were introduced in 1962 by Carl Adam Petri (PETRI, 1962). Since then, Petri-net theory has been extended in many ways and applied to many kinds of problems. Its popularity is due to both its easy-to-understand graphical representation of nets and its potential as a technique for formally analyzing concurrent systems. One way of classifying the large number of Petri-net formalisms that exist to date is to separate them into *colored* and *uncolored* formalisms. This thesis is mainly concerned with uncolored Petri nets. In this section, an introduction is given to the framework of Place/Transition nets, which is illustrative for any other uncolored Petri-net formalism. Good starting points for further reading on uncolored Petri nets are (PETERSON, 1981; REISIG, 1985; BEST AND FERNÁNDEZ C., 1988; MURATA, 1989). Colored Petri nets extend the basic Petri-net formalism with data, hierarchy, and time. A full treatment of colored Petri nets can be found in (JENSEN, 1992, 1995, 1997; VAN HEE, 1994).

The next subsection introduces the notion of bags, which play an important role in any Petri-net formalism. Section 2.3.2 presents the class of *labeled* P/T nets, which is the most general class of P/T nets considered in this thesis. Section 2.3.3 presents some theoretical results about P/T nets that are the basis for some of the analysis techniques available to analyze Petri-net models of concurrent systems. Section 2.3.4 introduces some subclasses of labeled P/T nets. The reason to consider subclasses of labeled P/T nets is that analysis techniques for these subclasses are generally more powerful than the techniques for labeled P/T nets. This is particularly true for the so-called class of free-choice P/T nets. Section 2.3.5 focuses on the analysis of free-choice P/T nets.

2.3.1 Notations for bags

In this thesis, bags are defined as finite multi-sets of elements from some alphabet A. A bag over alphabet A can be considered as a function from A to the natural numbers $\mathbb{N}$ such that only a finite number of elements from A is assigned a non-zero function value. For some bag X over alphabet A and $a \in A$, $X(a)$ denotes the number of occurrences of a in X, often called the cardinality of a in X. The set of all bags over A is denoted $\mathcal{B}(A)$. Note that any finite set of elements from A also denotes a unique bag over A, namely the function yielding 1 for every element in the set and 0 otherwise. The empty bag, which is the function yielding 0 for any element in A, is denoted $\mathbf{0}$. For the explicit enumeration of a bag, a notation similar to the notation for sets is used, but using square brackets instead of curly brackets and using superscripts to denote the cardinality of the elements. For example, $[a^2, b, c^3]$ denotes the bag with two elements a, one b, and three elements c; the bag $[a^2 \mid P(a)]$ contains two elements a for every a such that $P(a)$ holds, where P is some predicate on symbols of the alphabet under consideration. To denote individual elements of a bag, the same symbol "$\in$" is used as for sets: For any bag X over alphabet A and element $a \in A$, $a \in X$ if and only if $X(a) > 0$. The sum of two bags X and Y, denoted $X \uplus Y$, is defined as $[a^n \mid a \in A \wedge n = X(a) + Y(a)]$. The difference of X and Y, denoted $X - Y$, is defined as $[a^n \mid a \in A \wedge n = (X(a) - Y(a)) \max 0]$. The binding of sum and difference is left-associative. The restriction of X to some domain $D \subseteq A$, denoted $X \restriction D$, is defined as $[a^{X(a)} \mid a \in D]$. Restriction binds stronger than sum and difference. The notion of subbags is defined as expected: Bag X is a subbag of Y, denoted $X \leq Y$, if and only if for all $a \in A$, $X(a) \leq Y(a)$.

2.3.2 Labeled Place/Transition nets

Let U be some universe of identifiers; let L be some set of labels. The set of positive natural numbers is denoted $\mathbb{P}$.

Definition 2.3.1. (Labeled P/T net) An L-labeled Place/Transition net, or simply labeled P/T net, is a tuple (P, T, F, W, ℓ) where

i) $P \subseteq U$ is a finite set of *places*;

ii) $T \subseteq U$ is a finite set of *transitions* such that $P \cap T = \emptyset$;

iii) $F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs, called the *flow relation*;

iv) $W : F \to \mathbb{P}$ is a *weight function*;

v) $\ell : T \to L$ is a *labeling function*.

Let (P, T, F, W, ℓ) be a labeled P/T net. Elements of $P \cup T$ are referred to as *nodes*. A node $x \in P \cup T$ is called an *input node* of another node $y \in P \cup T$ if and only if there exists a directed arc from x to y; that is, if and only if $x F y$. Node x is called an *output node* of y if and only if there exists a directed arc from y to x. If x is a place in P, it is called an input place or an output place; if it is a transition, it is called an input or an output transition. The set of all input nodes of some node x is called the *preset* of x; its set of output nodes is called the *postset*. Two auxiliary functions $^\bullet\_, \_^\bullet : (P \cup T) \to \mathcal{P}(P \cup T)$ are defined that assign to each node its preset and postset, respectively. For any node $x \in P \cup T$, $^\bullet x = \{y \mid yFx\}$ and $x^\bullet = \{y \mid xFy\}$. Two additional functions $\boldsymbol{i}, \boldsymbol{o} : (P \cup T) \to \mathcal{B}(P \cup T)$ assign to each node its *bag* of input and output nodes, respectively. For any $x \in P \cup T$, $\boldsymbol{i}x = [y^{W(y,x)} \mid y \in {}^\bullet x]$ and $\boldsymbol{o}x = [y^{W(x,y)} \mid y \in x^\bullet]$.

Example 2.3.2. Besides a mathematical definition, labeled P/T nets also have an unambiguous graphical representation. Figure 2.3.3 on the following page shows a labeled P/T net representing a simple production unit. Places are depicted by circles, transitions by rectangles, and the flow relation by arcs. The small black dots residing in the places are called tokens, which are introduced below. The weight function is represented by labeling arcs with their weights; weights equal to one are omitted. Attached to each place is its identifier. Attached to each transition are its identifier and its label. In the example of Figure 2.3.3, it is assumed that the labeling function is the identity function.

A labeled P/T net as defined above is a static structure. However, labeled P/T nets also have a dynamic behavior. The dynamic behavior of a net is determined by its structure and its *state*. To express the state of a net, its places may contain *tokens*. In labeled P/T nets, tokens are nothing more than simple markers (see the graphical representation of a labeled P/T net in Figure 2.3.3 on the next page). The distribution of tokens over the places is often called the *marking* of the net.

Definition 2.3.4. (Marked, labeled P/T net) A *marked*, L-labeled P/T net is a pair (N, s), where $N = (P, T, F, W, \ell)$ is an L-labeled P/T net and where s is a bag over P denoting the marking of the net. The set of all marked, L-labeled P/T nets is denoted $\mathcal{N}$.

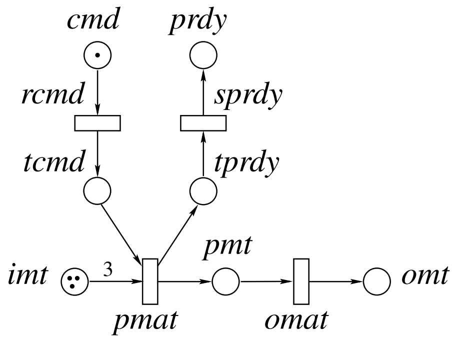


Figure 2.3.3: The graphical representation of a labeled P/T net.

The dynamic behavior of marked, labeled P/T nets is defined by a so-called *firing rule*, which is simply a transition relation defining the change in the state of a marked net when executing an action. To define the firing rule, it is necessary to formalize when a net is allowed to execute a certain action.

Definition 2.3.5. (Transition enabling) Let (N, s) be a marked, labeled P/T net in $\mathcal{N}$, where $N = (P, T, F, W, \ell)$. A transition $t \in T$ is *enabled*, denoted $(N, s)[t\rangle$, if and only if each of its input places p contains at least as many tokens as the cardinality of p in $\boldsymbol{i}t$. That is, $(N, s)[t\rangle \Leftrightarrow \boldsymbol{i}t \leq s$.

Example 2.3.6. In the marked net of Figure 2.3.3, transition *rcmd* is enabled. None of the other transitions is enabled.

When a transition t of a labeled P/T net is enabled, the net can *fire* this transition. Upon firing, t removes $W(p, t)$ tokens from each of its input places p; it adds $W(t, p)$ tokens to each of its output places p. This means that upon firing t, the marked net (N, s) changes into another marked net $(N, s - \boldsymbol{i}t \uplus \boldsymbol{o}t)$.

When firing a transition, the labeled P/T net executes an *action*. What action the net performs is determined by a so-called *action function*. Let A be some set of actions. For any labeled P/T net N equal to (P, T, F, W, ℓ), an action function $\alpha_N : T \to A$ is a function assigning actions to all transitions in T. The definition of the firing rule for labeled P/T nets given below is parameterized by an action function. Doing so yields a flexible definition which can be instantiated with a concrete action function when defining the semantics of nets. The parameterization of the firing rule differs from the usual definition of the firing rule (see, for example PETERSON, 1981; REISIG, 1985; MURATA, 1989). However, it proves to be useful in the remainder of this thesis.

Definition 2.3.7. (Firing rule) Let A be some set of actions; let, for any labeled P/T net $N = (P, T, F, W, \ell)$, the function $\alpha_N : T \to A$ be an action function. The firing rule $\_\, [\_\rangle\, \_ \subseteq \mathcal{N} \times A \times \mathcal{N}$ is the smallest relation satisfying for any (N, s) in $\mathcal{N}$, with $N = (P, T, F, W, \ell)$, and any $t \in T$,

$(N, s)[t\rangle \Rightarrow (N, s)\ [\alpha_N(t)\rangle\ (N, s - \boldsymbol{i}t \uplus \boldsymbol{o}t)$.

Tokens that are removed from the marking when firing a transition are often referred to as *consumed* tokens or the *consumption* of a transition; tokens that are added to the marking are referred to as *produced* tokens or the *production* of a transition.

There are many meaningful choices for concrete action functions. If labeled P/T nets are used to model programs written in some concurrent programming language where transition labels correspond to procedure identifications, actions could correspond to the implementations of the procedures. For theoretical purposes, the details of actions might not be relevant. Hence, one could choose the identity function on transitions as the action function, meaning that the action performed when firing a transition is the transition itself. Another feasible action function is the action function that assigns to each transition its label. In other words, the labels in a labeled P/T net are interpreted as actions.

Definition 2.3.8. (Labeling action function) Let $N = (P, T, F, W, \ell)$ be an L-labeled P/T net. The *labeling action function* $\alpha_N^l : T \to L$ is equal to the labeling function ℓ.

In the remainder of this section, the labeling action function is chosen as the standard action function for labeled P/T nets.

Example 2.3.9. In the labeled P/T net of Figure 2.3.3 on the preceding page, firing transition *rcmd* changes the marking from $[cmd,\ imt^3]$ to $[tcmd, imt^3]$, thus enabling transition *pmat*. Firing transition *pmat* yields the marking $[tprdy, pmt]$. This new marking enables both transitions *sprdy* and *omat*. These two transitions can be fired in arbitrary order, yielding the final marking $[prdy,\ omt]$. Since, in the net of Figure 2.3.3, the labeling function is simply the identity function, the choice of the labeling action function as the standard action function means that upon firing a transition t action t is performed. Note that the changes in the marking when firing a transition are independent of the chosen action function.

This example shows two clear advantages of labeled P/T nets over process spaces as a formalism to model concurrent systems. The representation of states by marked places allows to express in a very natural way that the production unit needs three pieces of input material to produce one piece of processed material. Such details are not expressible in process spaces in a straightforward way. Another advantage is that concurrency is represented in a very natural way. As opposed to the example in Figure 2.2.4 (d) on Page 7, it is not necessary to duplicate transitions *sprdy* and *omat*.

The definitions introduced so far suffice to give an operational semantics for marked, labeled P/T nets. In the remainder of this subsection, four possible semantics are given illustrating the flexibility of the framework of Section 2.2.

The usual Petri-net semantics (see, for example, PETERSON, 1981; REISIG, 1985; MURATA, 1989) does not distinguish successful termination and deadlock nor is there any difference between observable and silent behavior. In the framework developed in this chapter, such a semantics is defined as follows.

Definition 2.3.10. (Operational semantics) The operational semantics $\mathcal{S}(\mathcal{N})$ for L-labeled P/T nets without successful termination is the process space $(\mathcal{N}, L,\ [\,\rangle\ , \emptyset)$ with strong bisimilarity as the equivalence relation.

In order to distinguish successful termination from deadlock, a termination predicate is needed. A reasonable definition of successful termination in a P/T-net framework is to say that a labeled P/T net terminates successfully if and only if firing a transition results in the empty marking. If the marking of a net is not empty and it cannot fire any transitions anymore, the net is said to be in a deadlock. This definition for successful termination of labeled P/T nets was first given in (VAN GLABBEEK AND VAANDRAGER, 1987). Let $\downarrow$ be the set of all marked, labeled P/T nets $(N, \mathbf{0})$ in $\mathcal{N}$.

Definition 2.3.11. (Operational semantics) The operational semantics $\mathcal{S}^{\downarrow}(\mathcal{N})$ for L-labeled P/T nets with successful termination is the process space $(\mathcal{N}, L, [\,\rangle, \downarrow)$ with strong bisimilarity as the equivalence relation.

Example 2.3.12. Assuming the operational semantics $\mathcal{S}^{\downarrow}(\mathcal{N})$, the labeled P/T net of Figure 2.3.3 on Page 18 has a semantics which is almost identical to the process given in Figure 2.2.4 (d) on Page 7 (see also Example 2.3.9 on the previous page). Only its termination behavior is different. Since the final marking of the net of Figure 2.3.3 has tokens in *prdy* and *omt* (see again Example 2.3.9) and a marked net only terminates successfully if its marking is empty, the net of Figure 2.3.3 terminates in a deadlock. However, it is straightforward to define an operational semantics for labeled P/T nets in which the production unit terminates successfully. Let $(\mathcal{N}, L, [\,\rangle, \downarrow')$ be a process space where $\downarrow'$ is defined as the set of all $(N, [prdy, omt])$ in $\mathcal{N}$. It is easy to see that in an operational semantics based on this process space, the semantics of the production unit is identical to the process given in Figure 2.2.4 (d) on Page 7. Such an operational semantics could well be used to compare the (termination) behavior of different designs of a production unit. Of course, one could achieve the same results in semantics $\mathcal{S}^{\downarrow}(\mathcal{N})$ by simply removing places *prdy* and *omt* from the net structure.

In order to distinguish between observable and silent behavior, it suffices to assume that the set of labels L contains the special label τ. The labeling action function of Definition 2.3.8 on the previous page guarantees that firing a transition labeled τ results in a silent action.

Definition 2.3.13. (Operational semantics) The operational semantics $\mathcal{S}^{o}(\mathcal{N})$ for L-labeled P/T nets with silent actions but without successful termination is the process space $(\mathcal{N}, L, [\,\rangle, \emptyset)$ with branching bisimilarity as the equivalence relation.

Definition 2.3.14. (Operational semantics) The operational semantics $\mathcal{S}^{o\downarrow}(\mathcal{N})$ for L-labeled P/T nets with silent actions and successful termination is the process space $(\mathcal{N}, L, [\,\rangle, \downarrow)$, where $\downarrow$ is the termination predicate defined above, with branching bisimilarity as the equivalence relation.

2.3.3 Analysis of labeled P/T nets

P/T-net models of concurrent systems can be analyzed in many different ways. A commonly used technique to analyze the behavior of P/T nets is simulation. P/T nets are well suited for simulation techniques because of their intuitive graphical representation in combination with the precise definition of their dynamic behavior. To date, many tools for simulating Petri-net models exist. Three well-known tools are ExSpect (ASPT, 1994), Design/CPN (JENSEN ET AL., 1991), and PEP (BEST AND GRAHLMANN, 1995). All three tools support colored Petri nets. The Petri-net formalism which is the basis for ExSpect is described in (VAN HEE, 1994), whereas the formalism supported by Design/CPN can be found in (JENSEN, 1992). A good starting point for references to the various formalisms underlying PEP is (GRAHLMANN, 1997). In this thesis, simulation does not play an important role. A simple application of simulation techniques for Petri nets appears in Chapter 3. Simulation is not discussed in detail.

Other analysis techniques for Petri-net models are focused on properties of such models. The basic idea is that properties of a concurrent system can be phrased in terms of properties of its Petri-net model. This subsection presents a selection of properties that can be used to analyze labeled P/T nets. This selection is not exhaustive. However, it illustrates the analytical power of P/T nets and it is sufficient for the remainder of this thesis.

Some properties of P/T nets are defined on the structure of a net, whereas other properties concern their dynamic behavior. Therefore, assume that $(\mathcal{N}, L, [\,\rangle, \downarrow)$ is the process space in some operational semantics for labeled P/T nets, where the set of processes $\mathcal{N}$ is the class of marked, L-labeled P/T nets, $\_\,[\_\rangle\,\_ \subseteq \mathcal{N} \times L \times \mathcal{N}$ is the firing rule of Definition 2.3.7 on Page 18, instantiated with the labeling action function of Definition 2.3.8 on Page 19, and $\downarrow$ is some termination predicate on $\mathcal{N}$.

The first property defined in this subsection is a simple structural property which is often convenient in the analysis of P/T nets. The reflexive and transitive closure of a relation R is denoted R^*; the inverse of relation R is denoted R^{-1}.

Definition 2.3.15. (Connectedness) A labeled P/T net $N = (P, T, F, W, \ell)$ is *weakly connected*, or simply *connected*, if and only if, for every two nodes x and y in $P \cup T$, $x(F \cup F^{-1})^*y$. Net N is *strongly connected* if and only if, for every two nodes x and y in $P \cup T$, xF^*y.

Example 2.3.16. The P/T net in Figure 2.3.3 on Page 18 is connected, but not strongly connected.

The next definition is fundamental in the analysis of P/T nets. It defines the set of markings that a P/T net can reach from its initial marking. The set of reachable markings of a P/T net is also called the state space of the net. Recall that $\_ \stackrel{*}{\Longrightarrow} \_ \subseteq \mathcal{N} \times \mathcal{N}$ is the reachability relation defined in Definition 2.2.2 on Page 6.

Definition 2.3.17. (Reachable markings) The set of *reachable markings* of a marked, labeled P/T net $(N, s) \in \mathcal{N}$ with $N = (P, T, F, W, \ell)$, denoted $[N, s\rangle$, is defined as the set $\{s' \in \mathcal{B}(P) \mid (N, s) \stackrel{*}{\Longrightarrow} (N, s')\}$.

Sometimes it is convenient to know the sequence of transitions that are fired in order to reach some given marking. This thesis uses the following notations for sequences. Let A be some alphabet of identifiers. A *sequence of length n*, for some natural number $n \in \mathbb{N}$, over alphabet A is a function $\sigma : \{0, \ldots, n-1\} \rightarrow A$. The sequence of length zero is called the empty sequence and written ε. For the sake of readability, a sequence of positive length is usually written by juxtaposing the function values: For example, a sequence $\sigma = \{(0, a), (1, a), (2, b)\}$, for $a, b \in A$, is written aab. The set of all sequences of arbitrary length over alphabet A is written A^*.

Definition 2.3.18. (Firing sequence) Let (N, s) with $N = (P, T, F, W, \ell)$ be a marked, labeled P/T net in $\mathcal{N}$. A sequence $\sigma \in T^*$ is called a *firing sequence* of (N, s) if and only if for some natural number $n \in \mathbb{N}$, there exist markings $s_0, \ldots, s_n \in \mathcal{B}(P)$ and transitions $t_1, \ldots, t_n \in T$ such that $s = s_0$, $\sigma = t_1 \ldots t_n$ if $n > 0$ or $\sigma = \varepsilon$ if $n = 0$, and, for all i with $0 \leq i < n$, $(N, s_i)[t_{i+1}\rangle$ and $s_{i+1} = s_i - \boldsymbol{i}t_{i+1} \uplus \boldsymbol{o}t_{i+1}$. Sequence σ is said to be *enabled* in marking s, denoted $(N, s)[\sigma\rangle$. Firing the sequence σ results in the unique marking s_n, denoted $(N, s)\ [\sigma\rangle\ (N, s_n)$.

The following property is a direct result of the definitions given so far. It states that a marking of a labeled P/T net is reachable if and only if there is a firing sequence leading to that marking.

Property 2.3.19. *Let (N, s) with $N = (P, T, F, W, \ell)$ be a marked, labeled P/T net in $\mathcal{N}$. For any marking $s' \in \mathcal{B}(P)$, $s' \in [N, s\rangle$ if and only if there exists a firing sequence σ of (N, s) such that $(N, s)\ [\sigma\rangle\ (N, s')$.*

Proof. Definitions 2.3.17 (Reachable markings) and 2.3.18 (Firing sequence). □

Example 2.3.20. It is straightforward to verify that the set of reachable markings of the P/T net in Figure 2.3.3 on Page 18 equals $\{[cmd, imt^3], [tcmd, imt^3], [tprdy, pmt], [prdy, pmt], [tprdy, omt], [prdy, omt]\}$. Marking $[prdy, pmt]$ is the result of firing sequence *rcmd pmat sprdy*.

The following property is often useful when analyzing the set of reachable markings of a P/T net.

Property 2.3.21. (Monotonicity of reachable markings) *Let (N, s) with $N = (P, T, F, W, \ell)$ be a marked, labeled P/T net in $\mathcal{N}$. Let s' and s'' in $\mathcal{B}(P)$ be two markings of N. If s' is a reachable marking in $[N, s\rangle$, then marking $s' \uplus s''$ is reachable from $s \uplus s''$; that is, $s' \uplus s'' \in [N, s \uplus s''\rangle$.*

Proof. Let s' and s'' in $\mathcal{B}(P)$ be two markings of N such that s' is a reachable marking in $[N, s\rangle$. If $s' = s$, then the property is trivial. Therefore, assume $s' \neq s$. According to Property 2.3.19 and Definition 2.3.18 (Firing sequence), there exist a non-empty firing sequence $\sigma = t_1 \ldots t_n$, for some positive natural number $n \in \mathbb{P}$ and transitions $t_1, \ldots, t_n \in T$, and markings $s_0, \ldots, s_n \in \mathcal{B}(P)$ such that $s = s_0$, $s' = s_n$, and, for all i with $0 \leq i < n$, $(N, s_i)[t_{i+1}\rangle$ and $s_{i+1} = s_i - \boldsymbol{i}t_{i+1} \uplus \boldsymbol{o}t_{i+1}$. It follows easily from Definition 2.3.5 (Transition enabling) on Page 18 that, for all i with $0 \leq i < n$, $(N, s_i \uplus s'')[t_{i+1}\rangle$ and $s_{i+1} \uplus s'' = s_i \uplus s'' - \boldsymbol{i}t_{i+1} \uplus \boldsymbol{o}t_{i+1}$. Hence, Definition 2.3.18 yields that $(N, s \uplus s'')\ [\sigma\rangle\ (N, s' \uplus s'')$. Again Property 2.3.19 yields that $s' \uplus s'' \in [N, s \uplus s''\rangle$. □

Example 2.3.20 on the preceding page shows that it can be straightforward to calculate the set of reachable markings of a P/T net. However, in general, the state space of a non-trivial P/T net can be very large or even infinite. A P/T net with a finite state space is said to be bounded.

Definition 2.3.22. (Boundedness) A marked, labeled P/T net $(N, s) \in \mathcal{N}$ is *bounded* if and only if the set of reachable markings $[N, s\rangle$ is finite.

The following property characterizes boundedness; it is particularly useful to prove that a P/T net is unbounded.

Property 2.3.23. (Characterization of boundedness) *Let (N, s) be a marked, labeled P/T net in $\mathcal{N}$. Net (N, s) is bounded if and only if, for any markings $s' \in [N, s\rangle$ and $s'' \in [N, s'\rangle$, $s'' \geq s'$ implies $s'' = s'$.*

Proof. See (PETERSON, 1981, Section 4.2.1.1), where the property is formulated as a property of the coverability tree (called the reachability tree in (PETERSON, 1981)). The property is a direct result of Property 2.3.21 (Monotonicity of reachable markings) on the facing page and the fact that the coverability tree of a labeled P/T net is always finite. □

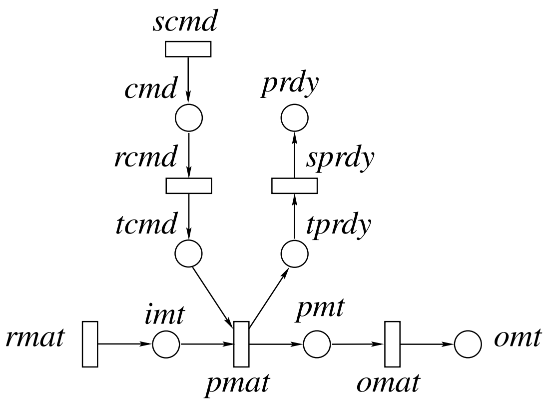


Figure 2.3.24: An example of a live and unbounded P/T net.

Example 2.3.25. The P/T net of Figure 2.3.3 on Page 18 is bounded. Example 2.3.20 on the preceding page shows that its state space is finite. Figure 2.3.24 shows a variant of the P/T net of Figure 2.3.3. It contains two new transitions, modeling an operator that sends commands to the production unit (*scmd*) and replenishes input material (*rmat*). The initial marking is the empty bag **0**. For convenience, all weights are set to one. As before, it is assumed that the labeling function is the identity function. It is not difficult to see that the net of Figure 2.3.24 is unbounded. Since the two abovementioned transitions have no input places, they are continuously enabled. Therefore, the number of tokens in places *cmd* and *imt*, and thus all other places, can increase indefinitely. Property 2.3.23 can be used to prove this claim formally. Firing transition *scmd* leads to marking [*cmd*], which is strictly larger than the initial marking **0**.

Another property of a P/T-net model, which is often meaningful from a design point of view, is that it contains no so-called dead transitions. A dead transition is a transition that is never able to fire.

Definition 2.3.26. (Dead transition) Let (N, s) be a marked, labeled P/T net in $\mathcal{N}$. A transition $t \in T$ is *dead* in (N, s) if and only if there is no reachable marking $s' \in [N, s\rangle$ such that $(N, s')[t\rangle$.

A property stronger than the absence of dead transitions is liveness. A P/T net is live if and only if, no matter what marking has been reached, it is always possible to enable an *arbitrary* transition of the net by firing any number of other transitions.

Definition 2.3.27. (Liveness) A marked, labeled P/T net $(N, s) \in \mathcal{N}$ with $N = (P, T, F, W, \ell)$ is *live* if and only if, for every reachable marking $s' \in [N, s\rangle$ and transition $t \in T$, there is a reachable marking $s'' \in [N, s'\rangle$ such that $(N, s'')[t\rangle$.

Property 2.3.28. *A live marked, labeled P/T net does not have any dead transitions.*

Proof. Definitions 2.3.26 and 2.3.27. □

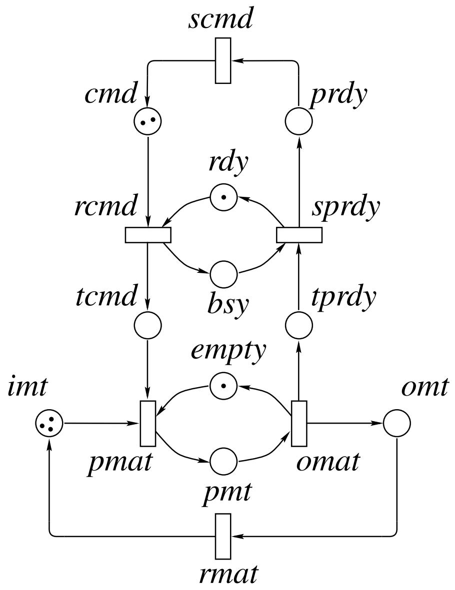


Figure 2.3.29: An example of a live and bounded P/T net.

Example 2.3.30. The P/T net of Figure 2.3.24 on the preceding page is live. Figure 2.3.29 shows another example of a live P/T net. It is a variant of the production unit of Figure 2.3.24. It contains three extra places. A token in place *rdy* means that the unit is ready to receive a command; a token in *bsy* means that it is busy processing; a token in *empty* models that the unit contains no material. Another difference with the unit of Figure 2.3.24 is that the processing-ready signal is not sent immediately after processing, but only after the output material has been delivered. A final difference concerns the commands and the input material for the unit. The operator only sends a new command to the unit after a processing-ready signal has been received. Material is only replenished after output material has been delivered. Initially, two commands have been sent to the unit and three pieces of input material are available.

It is possible to prove that the two P/T nets of Figures 2.3.24 and 2.3.29 are live by analyzing their state spaces. From such an analysis, it also follows that the net of Figure 2.3.29 is bounded, whereas we have already seen that the net of Figure 2.3.24 is unbounded.

The P/T net of Figure 2.3.3 on Page 18 is not live. Again, this follows easily from the state space (see Example 2.3.20 on Page 22).

A marking of a marked P/T net is called a home marking if and only if it is reachable from every marking reachable from the initial marking. Home markings play an important role in the analysis of termination behavior and iterative behavior. Consider a marked P/T net that terminates successfully when some given marking is reached. If this marking is a home marking, it means that the P/T net always has the option to terminate successfully. If the initial marking is a home marking of the net, then its behavior is iterative.

Definition 2.3.31. (Home marking) Let (N, s) be a marked, labeled P/T net in $\mathcal{N}$. A reachable marking $s' \in [N, s\rangle$ is a *home marking* of (N, s) if and only if, for any reachable marking $s'' \in [N, s\rangle$, $s' \in [N, s''\rangle$.

Example 2.3.32. The initial marking of the P/T net of Figure 2.3.29 on the preceding page is a home marking.

This subsection ends with two properties that a set of places of a labeled P/T net may exhibit. A set of places X of a P/T net is called a trap if every transition that needs a token from X to fire also returns a token to X. The preset and postset of a *set* of nodes X of a P/T net are defined as follows: $^\bullet X = (\cup x : x \in X : {}^\bullet x)$ and $X^\bullet = (\cup x : x \in X : x^\bullet)$.

Definition 2.3.33. (Trap) Let $N = (P, T, F, W, \ell)$ be a labeled P/T net. A set of places $X \subseteq P$ is called a *trap* if and only if $X^\bullet \subseteq {}^\bullet X$. A trap is *proper* if and only if it is not the empty set.

A characteristic property of a trap is that, once it becomes marked, it remains marked. Recall that, for some bag Y over alphabet A and element $a \in A$, $Y(a)$ denotes the number of occurrences of a in Y. For any $X \subseteq A$, $Y(X)$ denotes the total number of occurrences of elements of X in Y; that is, $Y(X) = (+a : a \in X : Y(a))$.

Property 2.3.34. *Let (N, s) be a labeled P/T net with $N = (P, T, F, W, \ell)$. Let $X \subseteq P$ be a trap of N. If $s(X) > 0$, then, for every reachable marking $s' \in [N, s\rangle$, $s'(X) > 0$.*

Proof. Definitions 2.3.33 (Trap), 2.3.17 (Reachable markings) on Page 21, and 2.3.7 (Firing rule) on Page 18. □

Example 2.3.35. Consider again the P/T net of Figure 2.3.29 on the preceding page. The following sets of places are traps: $\{rdy, bsy\}$, $\{empty, pmt\}$, $\{cmd, bsy, prdy\}$, $\{rdy, tcmd, pmt, tprdy\}$, $\{cmd, tcmd, pmt, tprdy, prdy\}$, and $\{imt, pmt, omt\}$. By definition, the union of any number of traps is again a trap. The P/T net of Figure 2.3.29 has no other proper traps than the six mentioned above and the ones that consist of the union of any number of these six traps.

A set of places X of a labeled P/T net is called a siphon if and only if every transition that puts a token in X upon firing also consumes a token from X.

Definition 2.3.36. (Siphon) Let $N = (P, T, F, W, \ell)$ be a labeled P/T net. A set of places $X \subseteq P$ is called a *siphon* if and only if ${}^\bullet X \subseteq X^\bullet$. A siphon is *proper* if and only if it is not the empty set.

A characteristic property of a siphon is that, once it becomes unmarked, it always remains unmarked.

Property 2.3.37. *Let (N, s) be a labeled P/T net with $N = (P, T, F, W, \ell)$. Let $X \subseteq P$ be a siphon of N. If $s(X) = 0$, then, for every reachable marking $s' \in [N, s\rangle$, $s'(X) = 0$.*

Proof. Definitions 2.3.36 (Siphon), 2.3.17 (Reachable markings) on Page 21, and 2.3.7 (Firing rule) on Page 18. □

Example 2.3.38. Consider again Figure 2.3.29 on Page 24. In this particular example, the set of siphons of the P/T net is identical to its set of traps given in Example 2.3.35 on the preceding page.

In this subsection, a non-exhaustive selection of properties of labeled P/T nets has been presented. However, some important aspects have not yet been addressed. Is it always possible to decide whether some marking of a labeled P/T net is reachable from its initial marking? Or whether it is a home marking? And is it always possible to decide whether some given labeled P/T net is bounded or live? The answer to all these questions is affirmative. However, the algorithms are very complex and inefficient. The reader interested in these algorithms, and in decidability and complexity results for P/T nets in general, is referred to the literature. A good starting point is (ESPARZA AND NIELSEN, 1994). A solution to improve the efficiency of analysis techniques for P/T nets is to consider *subclasses* of P/T nets.

2.3.4 Other classes of P/T nets

In this subsection, several subclasses of labeled P/T nets are defined by imposing restrictions on the net structure. A result of this approach is that the definitions and properties introduced in the previous subsection are valid for any of the subclasses of labeled P/T nets. Subclasses are presented in the order of decreasing expressive power.

The labeling function in labeled P/T nets is often only used for modeling purposes. It usually does not affect analysis techniques. Omitting transition labeling yields the class of unlabeled P/T nets. Recall that U is the universe of identifiers.

Definition 2.3.39. (P/T net) An *unlabeled* P/T net, or simply P/T net, is a U-labeled P/T net (P, T, F, W, ℓ), as defined in Definition 2.3.1 on Page 17, where the labeling function ℓ is the identity function. That is, for all $t \in T$, $\ell(t) = t$. An unlabeled P/T net is usually represented by the tuple (P, T, F, W).

Example 2.3.40. According to Definition 2.3.39 on the facing page, the labeled P/T nets of Figures 2.3.3 on Page 18, 2.3.24 on Page 23, and 2.3.29 on Page 24 are actually unlabeled P/T nets.

Definition 2.3.41. (Ordinary P/T net) An *ordinary* P/T net is a P/T net (P, T, F, W) as in Definition 2.3.39 on the preceding page with all weights equal to one. That is, for all $x \in F$, $W(x) = 1$. An ordinary P/T net is usually represented by the triple (P, T, F).

Example 2.3.42. The P/T net of Figure 2.3.3 is not ordinary. The weight of the arc between *imt* and *pmat* is greater than one. The P/T nets of Figures 2.3.24 and 2.3.29, on the other hand, are ordinary.

The following definition defines a particularly interesting class of ordinary P/T nets, so-called *free-choice* P/T nets. Free-choice P/T nets are characterized by the fact that two transitions sharing an input place always share all their input places. The class of free-choice P/T nets combines a reasonable expressive power with strong analysis techniques. Consequently, free-choice P/T nets have been extensively studied in the literature. The most important results on free-choice P/T nets have been brought together in (DESEL AND ESPARZA, 1995).

Definition 2.3.43. (Free-choice P/T net) A *free-choice* P/T net is an ordinary P/T net (P, T, F) as in Definition 2.3.41 such that, for all transitions $t, u \in T$, either $^\bullet t \cap {}^\bullet u = \emptyset$ or $^\bullet t = {}^\bullet u$.

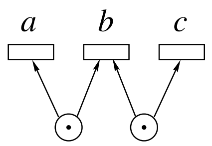


Figure 2.3.44: A non-free-choice construct: confusion.

Example 2.3.45. The two P/T nets of Figures 2.3.24 on Page 23 and 2.3.29 on Page 24 are both free-choice. Figure 2.3.44 shows a typical non-free-choice construct, called confusion. In the initial state, all three transitions a, b, and c are enabled. Transitions a and c are independent; they do not share any input places. However, both transitions compete for a token with transition b and are thus in conflict with b. The construct is called confusion, because firing, for example, transition a solves the conflict between the other two transitions b and c in favor of c. After a has fired only c is enabled.

Since free-choice P/T nets play such an important role in Petri-net theory, it is interesting to study some analysis techniques for this class. The next subsection is devoted to this topic.

The current subsection ends with two subclasses of free-choice P/T nets, called state machines and marked graphs. Their expressive power is limited, but they play a role in the analysis of other classes of P/T nets. An ordinary P/T net is a *state machine* if and only if every transition has exactly one input place and one output place.

Definition 2.3.46. (State machine) A *state machine* is an ordinary P/T net (P, T, F) as in Definition 2.3.41 on the preceding page such that, for every transition $t \in T$, $|^\bullet t| = 1$ and $|t^\bullet| = 1$.

An ordinary P/T net is a *marked graph* if and only if every place has exactly one input transition and one output transition.

Definition 2.3.47. (Marked graph) A *marked graph* is an ordinary P/T net (P, T, F) such that, for every place $p \in P$, $|^\bullet p| = 1$ and $|p^\bullet| = 1$.

Note that ordinary P/T nets, free-choice P/T nets, state machines, and marked graphs have all been defined as subclasses of *unlabeled* P/T nets. However, it is straightforward to extend any of these classes with transition labels, yielding for example labeled, ordinary P/T nets or labeled state machines.

2.3.5 Analysis of free-choice P/T nets

As mentioned in the previous subsection, free-choice P/T nets combine expressive power with strong analysis techniques. This subsection presents some results useful in the analysis of free-choice P/T nets. Except for Property 2.3.66 on Page 32, all results in this subsection appear in (DESEL AND ESPARZA, 1995). The main theorems are given without proof. Some other results are accompanied with proofs, because the proofs illustrate the use of the main theorems. The results in this subsection show that traps, siphons, and state machines play an important role in analyzing liveness, boundedness, and home markings of free-choice P/T nets.

As in Section 2.3.3, assume that $(\mathcal{N}, L, [\,\rangle\,, \downarrow)$ is the process space in some operational semantics for labeled P/T nets. The following definition is the basis for several results in this subsection. It defines the notion of a subnet of an ordinary P/T net.

Definition 2.3.48. (Subnet) Let $N = (P, T, F)$ be an ordinary P/T net, as defined in Definition 2.3.39 on Page 26; An ordinary P/T net $S = (P_0, T_0, F_0)$ with $P_0 \subseteq P$ and $T_0 \subseteq T$ is a *subnet* of N if and only if $F_0 = F \cap ((P_0 \times T_0) \cup (T_0 \times P_0))$. Subnet S is said to be generated by the set of nodes $P_0 \cup T_0$.

Example 2.3.50. Consider the P/T net of Figure 2.3.29 on Page 24. Figure 2.3.49 on the next page shows its subnet generated by the set of nodes {*cmd*, *rcmd*, *tcmd*, *pmat*, *pmt*, *omat*, *tprdy*, *sprdy*, *prdy*, *scmd*}.

The first main theorem of this subsection states that a connected free-choice P/T net is live if and only if every proper siphon includes an initially marked trap.

Theorem 2.3.51. (Commoner's theorem) *Let (N, s) with $N = (P, T, F)$ be a marked, connected free-choice P/T net in $\mathcal{N}$ such that T is not empty. Net (N, s) is live if and only if every proper siphon $X \subseteq P$ contains a trap $Y \subseteq X$ such that $s(Y) > 0$.*

Proof. (DESEL AND ESPARZA, 1995, Section 4.3) □

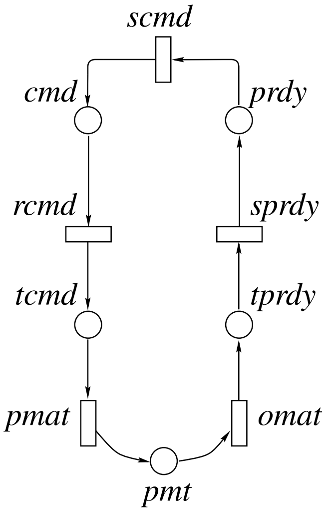


Figure 2.3.49: A subnet of the P/T net of Figure 2.3.29 on Page 24.

Example 2.3.52. Consider the free-choice P/T net of Figure 2.3.24 on Page 23. It is not difficult to see that this P/T net does not have any siphons. Hence, it trivially satisfies the condition of Theorem 2.3.51, which means it is live.

Figure 2.3.29 on Page 24 shows another free-choice P/T net. Its traps and siphons are given in Examples 2.3.35 on Page 25 and 2.3.38 on Page 26. It follows immediately that the net satisfies the condition of Commoner's theorem, which means that also this net is live. Note that these two results conform to the conclusions of Example 2.3.30 on Page 24.

The P/T net of Figure 2.3.3 on Page 18 is not free-choice, because one of its weights equals three. Therefore, consider a variant of this P/T net with all weights equal to one. Clearly, this variant is a free-choice net. The set consisting of the single place *cmd* is a siphon. Since it is not a trap, the condition of Commoner's theorem is not satisfied. As a result, the free-choice variant of the P/T net of Figure 2.3.3 is not live.

Theorem 2.3.51 (Commoner's theorem) on the facing page can be used to prove the following interesting property of live free-choice P/T nets. It says that a live free-choice P/T net remains live when an arbitrary number of tokens is added to its marking. The basic idea of the proof is that adding tokens to a marking cannot invalidate the requirements of Commoner's theorem.

Property 2.3.53. (Monotonicity of liveness) *Let (N, s) with $N = (P, T, F)$ be a live marked, free-choice P/T net in $\mathcal{N}$; let $s' \in \mathcal{B}(P)$ be a marking of N.The marked, free-choice P/T net $(N, s \uplus s')$ is live.*

Proof. Since Commoner's theorem is only valid for *connected* free-choice P/T nets, N is partitioned into connected subnets without any connections between them. Formally, let $N_0 = (P_0, T_0, F_0), \ldots, N_n = (P_n, T_n, F_n)$, for some $n \in \mathbb{N}$, be *connected* subnets of N such that, for all i with $0 \leq i \leq n$, N_i is generated by $P_i \cup T_i$. Furthermore, assume that P_0 through P_n, T_0 through T_n, and F_0 through F_n are partitionings of P, T, and F, respectively. Clearly, it follows from Definition 2.3.27 (Liveness) on Page 24 that, for any marking $s \in \mathcal{B}(P)$, (N, s) is live if and only if, for all i with $0 \le i \le n$, $(N_i, s \restriction P_i)$ is live.

Let N_i, for some i with $0 \le i \le n$, be an arbitrary subnet of N as defined above. To prove Property 2.3.53, it suffices to show that, for arbitrary markings $s, s' \in \mathcal{B}(P_i)$, $(N_i, s \uplus s')$ is live from the assumption that (N_i, s) is live. If $T_i = \emptyset$, Definition 2.3.27 (Liveness) immediately yields that $(N_i, s \uplus s')$ is live. Therefore, assume that $T_i \neq \emptyset$. Note that, since N is free-choice, also N_i is free-choice. Thus, it follows from the assumption that (N_i, s) is live and Theorem 2.3.51 (Commoner's theorem) on Page 28 that every proper siphon $X \subseteq P_i$ of N_i contains a trap $Y \subseteq X$ such that $s(Y) > 0$. Consequently, every proper siphon $X \subseteq P_i$ contains a trap $Y \subseteq X$ such that $(s \uplus s')(Y) > 0$. Again Theorem 2.3.51 yields that $(N_i, s \uplus s')$ is live. □

Note that liveness is not monotone for ordinary P/T nets. It is a nice exercise to find a live ordinary P/T net that is no longer live when one or more tokens are added to the initial marking. For the impatient reader, Figure 4.3.49 on Page 195 in Chapter 4 contains an example of such a net.

The following definition defines a very specific kind of subnet of ordinary P/T nets. It forms the basis for the second important theorem of this subsection.

Definition 2.3.54. (S-component) Let $N = (P, T, F)$ be an ordinary P/T net; let $^\bullet\_, \_^\bullet : P \cup T \to \mathcal{P}(P \cup T)$ be the preset and postset functions of N. The subnet $S = (P_0, T_0, F_0)$ of N with $P_0 \neq \emptyset$ is an *S-component* of N if and only if S is a strongly connected state machine, as defined by Definitions 2.3.15 on Page 21 and 2.3.46 on Page 28, such that, for every place $p \in P_0$, $^\bullet p \cup p^\bullet \subseteq T_0$.

Example 2.3.55. The ordinary P/T net of Figure 2.3.49 on the preceding page is an S-component of the P/T net of Figure 2.3.29 on Page 24.

A characteristic property of an S-component of some ordinary P/T net is that the number of tokens in the places of the S-component is constant for all reachable markings.

Property 2.3.56. *Let (N, s) be a marked, ordinary P/T net in $\mathcal{N}$; let (P, T, F) be an S-component of N. For any reachable marking $s' \in [N, s\rangle$, $s'(P) = s(P)$.*

Proof. Induction on the length of any firing sequence needed to reach marking s' from marking s plus Property 2.3.19 on Page 22. □

Example 2.3.57. Consider again the P/T net of Figure 2.3.29 on Page 24 and its S-component of Figure 2.3.49 on the previous page. It is not difficult to see that the number of tokens in the S-component is two for every reachable marking.

Property 2.3.56 has many interesting consequences. The following property states that the marking of a live connected ordinary P/T net marks all of its S-components.

Property 2.3.58. *Let (N, s) with $N = (P, T, F)$ be a live marked, connected ordinary P/T net in $\mathcal{N}$ such that T is not empty. For every S-component (P_0, T_0, F_0) of N, $s(P_0) > 0$.*

Proof. If N has no places the property is trivial. Therefore, assume that P is not empty. Let (P_0, T_0, F_0) be an S-component of N such that P_0 is not empty. Assume that $s(P_0) = 0$. It follows from the assumption that T is not empty and Definitions 2.3.15 (Connectedness) on Page 21 and 2.3.54 (S-component) on the preceding page that T_0 is not empty. Property 2.3.56 implies that any of the places in P_0 is unmarked in any marking reachable from s. Thus, it follows from Definitions 2.3.26 (Dead transition) on Page 24 and again 2.3.54 (S-component) that all transitions in T_0 are dead in (N, s). However, this contradicts Property 2.3.28 on Page 24, which states that a live P/T net cannot have dead transitions. Hence, $s(P_0) > 0$. □

Property 2.3.58 can be strengthened for a certain class of free-choice P/T nets.

Property 2.3.59. *Let (N, s) with $N = (P, T, F)$ be a live and bounded marked, connected free-choice P/T net such that T is not empty. Let $s' \in \mathcal{B}(P)$ be a marking of N. The marked P/T net (N, s') is live if and only if, for every S-component (P_0, T_0, F_0) of N, $s'(P_0) > 0$. That is, (N, s') is live if and only if s' marks every S-component of N.*

Proof. The implication from left to right follows immediately from Property 2.3.58 on the facing page. The other implication is a consequence of Theorem 2.3.51 (Commoner's theorem) on Page 28 and several results concerning siphons and traps, see (DESEL AND ESPARZA, 1995, Section 5.2) for details. □

An S-cover of an ordinary P/T net is a set of S-components such that each place of the net is contained by at least one S-component.

Definition 2.3.60. (S-cover) Let $N = (P, T, F)$ be an ordinary P/T net; let C be a set of S-components of N. Set C is called an *S-cover* of N if and only if, for every place $p \in P$, there exists an S-component (P_0, T_0, F_0) in C such that $p \in P_0$.

The second important theorem of this subsection is the S-coverability theorem.

Theorem 2.3.61. (S-coverability theorem) *Let (N, s) be a live and bounded marked, free-choice P/T net. Net N has an S-cover.*

Proof. (DESEL AND ESPARZA, 1995, Section 5.1) □

Example 2.3.62. In Example 2.3.30 on Page 24, it was argued that the P/T net of Figure 2.3.29 also on Page 24 is live and bounded. Since this P/T net is free-choice, it must have an S-cover. It is not difficult to verify that the four subnets generated by the sets of nodes {*cmd*, *rcmd*, *tcmd*, *pmat*, *pmt*, *omat*, *tprdy*, *sprdy*, *prdy*, *scmd*}, {*rdy*, *rcmd*, *bsy*, *sprdy*}, {*empty*, *pmat*, *pmt*, *omat*}, and {*imt*, *pmat*, *pmt*, *omat*, *omt*, *rmat*} are S-components that form an S-cover. The first one of these S-components is the one shown in Figure 2.3.49 on Page 29. Note that the marking in Figure 2.3.29 marks all S-components, which conforms to Property 2.3.59.

The S-coverability theorem can be used to prove, for example, the following property of live and bounded free-choice P/T nets.

Property 2.3.63. *Let (N, s) with $N = (P, T, F)$ be a live and bounded marked, free-choice P/T net in $\mathcal{N}$. For every marking $s' \in \mathcal{B}(P)$, net (N, s') is bounded.*

Proof. According to Definition 2.3.22 (Boundedness) on Page 23, it must be shown that the set of reachable markings $[N, s'\rangle$ is finite. First, note that the number of tokens in marking s', $s'(P)$, is finite. Second, it follows from Theorem 2.3.61 (S-coverability theorem) on the preceding page that N has an S-cover C. Since the number of places of a P/T net is finite, it follows from Definition 2.3.60 (S-cover) on the previous page that the number of S-components in C, $|C|$, is also finite. Since every S-component initially contains at most $s'(P)$ tokens, it follows from Property 2.3.56 on Page 30 that every reachable marking $s'' \in [N, s'\rangle$ contains at most $s'(P) \cdot |C|$ tokens; that is, for every $s'' \in [N, s'\rangle$, $s''(P) \leq s'(P) \cdot |C|$. Since the number of places of N is finite, it follows that $[N, s'\rangle$ is finite. □

The basis for the S-coverability theorem is the notion of an S-component. S-components are defined in terms of state machines, as introduced in the previous subsection. In a similar way, marked graphs form the basis of so-called T-components. Analogously to the S-coverability theorem, it can be shown that live and bounded free-choice P/T nets are covered by T-components. See (DESEL AND ESPARZA, 1995) for more details.

The third and final theorem of this subsection states that a reachable marking of a live and bounded connected free-choice P/T net is a home marking if and only if it marks every proper trap of the net.

Theorem 2.3.64. (Home-marking theorem) *Let (N, s) with $N = (P, T, F)$ be a live and bounded marked, connected free-choice P/T net in $\mathcal{N}$ such that T is not empty. A reachable marking $s' \in [N, s\rangle$ is a home marking of (N, s) if and only if, for every proper trap $X \subseteq P$, $s'(X) > 0$.*

Proof. (DESEL AND ESPARZA, 1995, Section 8.2) □

Example 2.3.65. Consider again the P/T net of Figure 2.3.29 on Page 24 and its traps given in Example 2.3.35 on Page 25. Since the initial marking marks every proper trap of the net, it is a home marking, which conforms to the claim of Example 2.3.32 on Page 25.

The Home-marking theorem can be used to prove monotonicity of home markings for live and bounded free-choice P/T nets.

Property 2.3.66. (Monotonicity of home markings) *Let (N, s) with $N = (P, T, F)$ be a live and bounded marked, free-choice P/T net in $\mathcal{N}$. Let s' and s'' in $\mathcal{B}(P)$ be two markings of N. If s' is a home marking of (N, s), then $s' \uplus s''$ is a home marking of $(N, s \uplus s'')$.*

Proof. If N has no transitions, the property is trivial. Therefore assume that T is not empty. Without loss of generality, it may also be assumed that N is connected. (If N is not connected, it is possible to consider the partitioning of connected subnets of N, similar to the proof of Property 2.3.53 (Monotonicity of liveness) on Page 29.)

Assume that s' is a home marking of (N, s). The goal is to use the Home-marking theorem to prove that $s' \uplus s''$ is a home marking of $(N, s \uplus s'')$. Thus, it must be shown that $(N, s \uplus s'')$ is live and bounded, that $s' \uplus s'' \in [N, s \uplus s'')$, and that $s' \uplus s''$ marks every proper trap of N.

Since (N, s) is live and bounded, Properties 2.3.53 on Page 29 and 2.3.63 on the facing page yield that $(N, s \uplus s'')$ is live and bounded. Since s' is a home marking of (N, s), it follows from Definition 2.3.31 (Home marking) on Page 25 and Property 2.3.21 on Page 22 that $s' \uplus s'' \in [N, s \uplus s'')$. It follows from the Home-marking theorem that s' marks every proper trap of N and, thus, that also $s' \uplus s''$ marks every proper trap of N. As a result, the conditions of the Home-marking theorem are satisfied proving that $s' \uplus s''$ is a home marking of $(N, s \uplus s'')$. □

Similar to Property 2.3.53 (Monotonicity of liveness) on Page 29, Property 2.3.66 does not generalize to ordinary P/T nets. Figure 4.3.49 on Page 195 shows a live and bounded ordinary P/T net with a home marking that does not satisfy the monotonicity requirement stated in Property 2.3.66.

The three main theorems of this subsection, Commoner's theorem, the S-coverability theorem, and the home-marking theorem, form the basis for several efficient analysis techniques for free-choice P/T nets. None of the results generalizes to ordinary P/T nets, although some of them (partially) generalize to subclasses of ordinary P/T nets that are slightly larger than free-choice P/T nets. The interested reader is referred to (DESEL AND ESPARZA, 1995, Chapter 10).

2.4 Process Algebra

This section gives an introduction to process algebra in the style of the Algebra of Communicating Processes, abbreviated ACP, (see BERGSTRA AND KLOP, 1984; BAETEN AND VERHOEF, 1995; BAETEN AND WEIJLAND, 1990). Whereas Petri-net formalisms are *graphical* by nature, process algebras are *symbolic*. Process algebras are often used for the analysis of concurrent-system behavior, in particular, the *comparison* of behaviors. There are many different kinds of process algebras. Apart from the already mentioned ACP, the process algebras CCS (MILNER, 1980, 1989) and CSP (HOARE, 1985) are the most well-known. Specifications and verifications in ACP are based on an equational style of reasoning, as opposed to CCS and CSP, which are model-based. In the latter case, the starting point of the process algebra is an operational semantics. The goal is to find equational laws that are valid in the semantics. In the former case, the starting point is an equational theory, which may have several operational (or non-operational) semantic interpretations. For a detailed comparison of ACP, CCS, and CSP, the reader is referred to (BAETEN AND WEIJLAND, 1990, Chapter 8).

The next subsection introduces the process algebra BPA which can be used to reason about sequential processes only, but which shows the essentials of any process-algebraic theory. Section 2.4.2 extends BPA in such a way that it is possible to reason about deadlocks in a process. In Section 2.4.3, concurrency in process algebra is introduced. Section 2.4.4 presents a process algebra for reasoning about processes with silent behavior.

Finally, Section 2.4.5 shows how to formalize a simple form of recursion in a process-algebraic setting.

2.4.1 Basic Process Algebra

The most elementary process algebra in the style of ACP is the so-called Basic Process Algebra, abbreviated BPA. It can be used to reason about the behavior of *sequential* processes. As mentioned, any ACP-style process algebra is basically an *equational theory*. An equational theory consists of a *signature* and a set of *axioms*. The signature defines the *sorts* of the theory, a set of *variables* for each sort, and the *functions* of the theory. Functions and variables can be used to construct *terms*. In process algebra, terms usually represent processes. Terms not containing any variables are called *closed terms*. The axioms of the theory determine which terms are equal. A 0-ary function is often called a *constant*; other functions are often called *operators*.

The equational theory BPA. The signature and axioms of BPA are given in Table 2.4.1. The theory is parameterized by a set of constants $\boldsymbol{A}$, which is a set of actions. The first part of Table 2.4.1 lists the sorts of BPA; the second part defines the operators and the third part gives the variables and lists the axioms. Note that the signature of BPA has no variables of sort $\boldsymbol{A}$. Also note that new variables may be introduced any time when necessary. An informal explanation of the operators and the axioms is given below.

BPA($\boldsymbol{A}$)	
$\boldsymbol{P}$; $\boldsymbol{A} \subseteq \boldsymbol{P}$	
$\_ + \_, \_ \cdot \_ : \boldsymbol{P} \times \boldsymbol{P} \to \boldsymbol{P}$	
$x, y, z : \boldsymbol{P}$;	
$x + y = y + x$	$A1$
$(x + y) + z = x + (y + z)$	$A2$
$x + x = x$	$A3$
$(x + y) \cdot z = x \cdot z + y \cdot z$	$A4$
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	$A5$

Table 2.4.1: The process algebra BPA.

As mentioned, $\boldsymbol{A}$ is a set of actions. Terms of sort $\boldsymbol{P}$ represent processes. Each action is a process, namely the process that can only execute the action and then terminates. The operators $+$ and $\cdot$ denote choice and sequential composition, respectively. Sequential composition binds stronger than choice. Given some processes, the operators of BPA can be used to construct new processes. If two processes x and y are composed using the choice operator, the behavior of the new process $x + y$ may be either x or y. Process $x \cdot y$ first performs x and, upon successful termination of x, it performs y. Thus, choice and sequential composition are two elementary operators in describing the behavior of sequential processes. Most of the axioms of BPA are self-explanatory. Only Axiom $A4$ might need some explanation. It states the *right distributivity* of sequential composition over choice. The converse, left distributivity, is not an axiom of the theory. As a result, processes with different moments of choice are distinguished. Example 2.2.12 on Page 10 explains why this can be useful.

An equational theory like BPA defines an equivalence relation, namely *derivability* from the axioms of the theory. For any terms x and y in some given process algebra X, $X \vdash x = y$ denotes that $x = y$ can be derived from the axioms of X. Derivability is defined as follows. First, the axioms themselves can be derived from the axioms of an equational theory. Second, since derivability is an equivalence relation, it is reflexive, symmetric, and transitive. Third, if an equation is derivable from the axioms, then also any equation obtained by substituting terms for variables in this equation is derivable. Finally, any equation obtained by replacing a term in an arbitrary context by another derivably equivalent term is also derivable from the theory. The axioms of an equational theory must be chosen in such a way that derivability defines a meaningful equivalence relation on processes. Below, it is explained that, in the case of the theory BPA, derivability corresponds to bisimilarity.

Example 2.4.2. It can be shown that BPA $\vdash rcmd \cdot ((pmat_1 + pmat_2) \cdot omat) = rcmd \cdot (pmat_1 \cdot omat + pmat_2 \cdot omat)$. The first step is to substitute $pmat_1$, $pmat_2$, and $omat$ for the variables x, y, and z in Axiom $A4$, which shows that BPA $\vdash t_1 = t_2$, where t_1 is the term $(pmat_1 + pmat_2) \cdot omat$ and t_2 denotes the term $pmat_1 \cdot omat + pmat_2 \cdot omat$. The second and final step consists of an application of the context rule explained above: Replacing term t_1 in $rcmd \cdot t_1$ with the equivalent term t_2 yields the desired result.

The intuitive meaning of the operators, the axioms, and the induced equivalence relation given above can be formalized by giving an operational semantics. However, before going into more details about the operational semantics for BPA, a few more aspects of the theory BPA itself are explained.

The main purpose of a theory as BPA is to reason about processes in an equational way. The goal is to prove a desired equality on processes by applying simple term-rewriting techniques. As a consequence, a very useful property of a process algebra is when its (closed) terms can be reduced to (unique) normal forms. If such normal forms exist, an equational proof becomes very simple. The equality of two process terms can be shown by reducing them to their normal forms. If the normal forms are equal, then, obviously, also the two processes are equal. By considering Axioms $A3$, 4, and 5 in Table 2.4.1 on the facing page as rewrite rules from left to right, it can be shown that closed BPA terms have normal forms. The proof and more details about term rewriting in the context of process algebra can be found in (BAETEN AND VERHOEF, 1995). For the purpose of this thesis, it is sufficient to know that the normal forms of closed BPA terms are contained in a very specific class of terms, called *basic terms*.

Definition 2.4.3. (Basic terms) The set of basic BPA terms, denoted $\mathcal{B}(\text{BPA})$, is inductively defined as follows. The actions $\boldsymbol{A}$ are contained in $\mathcal{B}(\text{BPA})$. Furthermore, for any $a \in \boldsymbol{A}$ and $s, t \in \mathcal{B}(\text{BPA})$, also $a \cdot t$ and $s + t$ are elements of $\mathcal{B}(\text{BPA})$.

Note that not all basic terms are normal forms of closed terms. For example, basic term $a+a$, where a is some action in $\boldsymbol{A}$, can be reduced by means of Axiom $A3$ and is, therefore, not a normal form.

The set of all closed terms of X is denoted $\mathcal{C}(X)$.

Property 2.4.4. (Elimination) *For any closed term* $p \in \mathcal{C}(\mathrm{BPA})$, *there exists a basic term* $t \in \mathcal{B}(\mathrm{BPA})$, *such that* $\mathrm{BPA} \vdash p = t$.

Proof. It is straightforward to prove the property using Axioms $A3$, 4, and 5 in Table 2.4.1 on Page 34 as rewrite rules from left to right. For details, see (Baeten and Verhoef, 1995). □

The above result is called the *elimination* property, because it shows that the general sequential composition $\cdot$ as defined in Table 2.4.1 on Page 34 can be eliminated from any closed term, yielding a term containing only so-called *prefix compositions*. A prefix composition is a sequential composition whose left operand is a single action (see Definition 2.4.3 (Basic terms) on the previous page). The elimination property is useful in inductive proofs in process algebra (see Section 2.4.3).

An operational semantics for BPA. The processes in the operational semantics for BPA are the *closed* BPA terms. It remains to give a transition relation and a termination predicate. The theory BPA does not allow to distinguish between successful termination and deadlock. In Section 2.4.2, BPA is extended in such a way that it is possible to make this distinction. The usual semantics for BPA is that termination means successful termination. For this purpose, a special process $\surd$, pronounced "tick," is introduced, which is the only process that can terminate successfully. The transition relation $\_ \stackrel{-}{\longrightarrow} \_ \subseteq (\mathcal{C}(\mathrm{BPA}) \cup \{\surd\}) \times \boldsymbol{A} \times (\mathcal{C}(\mathrm{BPA}) \cup \{\surd\})$ can now be defined as the smallest relation satisfying the derivation rules in Table 2.4.5. The rules conform to the intuitive explanation given above.

$a : \boldsymbol{A};\ p, p', q, q' : \mathcal{C}(\mathrm{BPA});$

$$a \xrightarrow{a} \surd \qquad \frac{p \xrightarrow{a} p'}{p \cdot q \xrightarrow{a} p' \cdot q} \qquad \frac{p \xrightarrow{a} \surd}{p \cdot q \xrightarrow{a} q}$$

$$\frac{p \xrightarrow{a} p'}{p + q \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} q'}{p + q \xrightarrow{a} q'} \qquad \frac{p \xrightarrow{a} \surd}{p + q \xrightarrow{a} \surd} \qquad \frac{q \xrightarrow{a} \surd}{p + q \xrightarrow{a} \surd}$$

Table 2.4.5: The transition relation for BPA.

Definition 2.4.6. (Operational semantics) The operational semantics $\mathcal{S}(\mathrm{BPA})$ is the process space $(\mathcal{C}(\mathrm{BPA}) \cup \{\surd\}, \boldsymbol{A}, \longrightarrow, \{\surd\})$, where $\longrightarrow$ is the transition relation defined above, with strong bisimilarity as the equivalence relation.

Example 2.4.7. Consider Example 2.2.5 on Page 6. The process depicted in Figure 2.2.4 (a) on Page 7 can be specified in BPA by the closed term $rcmd \cdot pmat \cdot omat$. Since sequential composition is associative, brackets are omitted. It is easy to see that the semantics of this closed term, assuming the operational semantics $\mathcal{S}$(BPA), is indeed the desired process. The two processes depicted in Figure 2.2.4(b) and (c) cannot be expressed in BPA. The theory has no means to specify deadlocks nor does it allow iterative or other recursive specifications. Although BPA has no explicit concurrency operator, it is possible to specify the process of Figure 2.2.4(d) as follows: $rcmd \cdot pmat \cdot (omat \cdot sprdy + sprdy \cdot omat)$.

Example 2.4.8. Consider Example 2.2.12 on Page 10. In BPA, the three processes depicted in Figure 2.2.11 on Page 9 are defined as $(rcmd \cdot pmat_1 + rcmd \cdot pmat_2) \cdot omat$, $rcmd \cdot (pmat_1 + pmat_2) \cdot omat$, and $rcmd \cdot (pmat_1 \cdot omat + pmat_2 \cdot omat)$. The first two terms cannot be proved equal in BPA, because the theory lacks an axiom stating the left distributivity of sequential composition over choice. The second and third process, on the other hand, are equal in BPA. As shown in Example 2.4.2 on Page 35, the desired equality follows from a single application of Axiom $A4$, stating the right distributivity of sequential composition over choice.

Not every operational semantics is suitable for an equational theory such as BPA. The minimal requirement for a proper semantics is that the axioms of the theory must be *sound* for the semantics. A set of axioms is sound for a semantics only if any equality that can be derived from the axioms is also *valid* in the semantics. An equality between processes is valid in a semantics if and only if the processes are equivalent according to the semantic equivalence. Validity is formally defined as follows. Let p and q be two closed terms in some given process algebra X. For any operational semantics $\mathcal{S}$ for X, the equation $p = q$ is valid in $\mathcal{S}$, denoted $\mathcal{S} \models p = q$, if and only if $p \approx q$, where $\approx$ is the equivalence in $\mathcal{S}$. The soundness requirement implies that the equivalence relation must be a *congruence* for all the operators in the algebra. That is, for a given semantic equivalence $\approx$ the following property must be satisfied. Let f be any n-ary function in the signature of the process algebra, where n is some natural number; let $p_1, \ldots, p_n, q_1, \ldots, q_n$ be closed terms of the appropriate sorts such that $p_1 \approx q_1, \ldots, p_n \approx q_n$. Then, the congruence property requires that $f(p_1, \ldots, p_n) \approx f(q_1, \ldots, q_n)$.

Example 2.4.9. A simple example shows why the congruence property is necessary. Assume that $\approx$ is a semantic equivalence in an operational semantics for which the axioms of BPA are sound. Furthermore, assume that $\approx$ is not a congruence for the choice operator. This means that there exist closed BPA terms p_1, p_2, q_1, and q_2 satisfying $i)$ BPA $\vdash p_1 = q_1$ and BPA $\vdash p_2 = q_2$, and $ii)$ $p_1 \approx q_1$ and $p_2 \approx q_2$, but not satisfying $p_1 + p_2 \approx q_1 + q_2$. Since BPA $\vdash p_1 + p_2 = q_1 + q_2$, this clearly contradicts the soundness of BPA for the semantics.

Property 2.4.10. (Congruence) *Strong bisimilarity, $\sim$, is a congruence for the operators of* BPA.

Proof. See (BAETEN AND VERHOEF, 1995). Note that the proof in (BAETEN AND VERHOEF, 1995) uses the fact that the derivation rules used to define the operational semantics for BPA are in the so-called *path* format. Although the formulation of the operational semantics in this section follows a slightly different notation, it is straightforward to give an equivalent definition satisfying the desired format. □

Besides soundness, a useful property of a set of axioms of an equational theory is *completeness*. A set of axioms is complete if every equality that is valid in the semantics is also derivable from the axioms. The following theorem is a standard result in process algebra.

Theorem 2.4.11. (Soundness and completeness) *For any closed terms* $p, q \in \mathcal{C}(\mathrm{BPA})$, $\mathrm{BPA} \vdash p = q \Leftrightarrow \mathcal{S}(\mathrm{BPA}) \models p = q$.

Proof. (BAETEN AND VERHOEF, 1995) □

This theorem means that the semantics $\mathcal{S}(\mathrm{BPA})$ defined in Definition 2.4.6 on Page 36 captures all the essential aspects of BPA, as far as closed terms are considered. A consequence of the soundness result and the fact that bisimilarity preserves the branching structure of processes is that derivability also preserves the branching structure of processes, as was already mentioned earlier.

2.4.2 Basic Process Algebra with inaction

As explained in the previous subsection, BPA does not allow to distinguish between successful termination and deadlock. Therefore, in this subsection, BPA is extended to BPA_δ, which is an abbreviation of Basic Process Algebra with *inaction*.

The equational theory BPA_δ. Table 2.4.12 shows the equational theory BPA_δ. The first entry means that BPA_δ is an extension of BPA. The other entries of Table 2.4.12 have a meaning as explained before.

$\mathrm{BPA}_\delta(\boldsymbol{A})$	
$\mathrm{BPA}(\boldsymbol{A})$	
$\delta : \boldsymbol{P}$	
$x : \boldsymbol{P}$;	
$x + \delta = x$	$A6$
$\delta \cdot x = \delta$	$A7$

Table 2.4.12: The process algebra BPA_δ.

In BPA_δ, one new process is introduced, namely δ. Process δ stands for *inaction*, often also called *deadlock*. However, the former name is best suited, as follows from Axiom $A6$. It says that a process which can choose between some behavior x and doing nothing is equivalent to the process that has no choice and can only do x. Hence, in the context of a choice, δ is not a true deadlock. Axiom $A7$ shows that δ is a deadlock in the context of a sequential composition.

As before, BPA_δ terms can be reduced to basic terms. However, the set of basic BPA_δ terms differs from the set of basic BPA terms because δ is a basic BPA_δ term (and can therefore be used in the construction of other basic BPA_δ terms).

Definition 2.4.13. (Basic terms) The set of basic BPA_δ terms, denoted $\mathcal{B}(\text{BPA}_\delta)$, is inductively defined as follows. The actions $\boldsymbol{A}$ and process δ are contained in $\mathcal{B}(\text{BPA}_\delta)$. Furthermore, for any $a \in \boldsymbol{A}$ and $s, t \in \mathcal{B}(\text{BPA}_\delta)$, also $a \cdot t$ and $s + t$ are elements of $\mathcal{B}(\text{BPA}_\delta)$.

Property 2.4.14. (Elimination) *For any closed term* $p \in \mathcal{C}(\text{BPA}_\delta)$, *there is a basic term* $t \in \mathcal{B}(\text{BPA}_\delta)$, *such that* $\text{BPA}_\delta \vdash p = t$.

Proof. (BAETEN AND VERHOEF, 1995) □

An operational semantics for BPA_δ. The processes in the operational semantics for BPA_δ are the closed BPA_δ terms. As before, a special process $\surd$ is introduced, which is the only process that can terminate successfully. The transition relation is the same as the transition relation for BPA. That is, $\_ \xrightarrow{\_} \_ \subseteq (\mathcal{C}(\text{BPA}_\delta) \cup \{\surd\}) \times \boldsymbol{A} \times (\mathcal{C}(\text{BPA}_\delta) \cup \{\surd\})$ is the smallest relation satisfying the derivation rules in Table 2.4.5 on Page 36, where $p, p', q,$ and q' range over $\mathcal{C}(\text{BPA}_\delta)$. This means that the new process δ cannot perform any actions, which conforms to the claim that it represents inaction. As a result, it is possible to distinguish between successful termination and deadlock. Note that the process that can only terminate successfully *cannot* be expressed in the theory BPA_δ. It is possible to extend BPA or BPA_δ with a constant called the *empty process*. The empty process is exactly the process that can only terminate successfully. For more details about the empty process, see (BAETEN AND VERHOEF, 1995; BAETEN AND WEIJLAND, 1990).

Definition 2.4.15. (Operational semantics) The operational semantics $\boldsymbol{\mathcal{S}}(\text{BPA}_\delta)$ is the process space $(\mathcal{C}(\text{BPA}_\delta) \cup \{\surd\}, \boldsymbol{A}, \longrightarrow, \{\surd\})$ with strong bisimilarity as the equivalence relation.

Example 2.4.16. Consider again Example 2.2.5 on Page 6. Since BPA_δ is an extension of BPA, the BPA_δ specifications of the processes in Figure 2.2.4 (a) and (d) on Page 7 are identical to the BPA specifications given in Example 2.4.7 on Page 37. In BPA_δ, it is possible to define the process depicted in Figure 2.2.4(b) as follows: $rcmd \cdot \delta + rcmd \cdot pmat \cdot omat$. Recall that it is not possible to define this process in BPA (see Example 2.4.7). It is still not possible to define the iterative process of Figure 2.2.4(c).

As before, strong bisimilarity is a congruence for the operators of BPA_δ and the axioms of BPA_δ are sound and complete for the operational semantics.

Property 2.4.17. (Congruence) *Strong bisimilarity,* $\sim$, *is a congruence for the operators of* BPA_δ.

Proof. (BAETEN AND VERHOEF, 1995) □

Theorem 2.4.18. (Soundness and completeness) *For any closed terms* $p, q \in \mathcal{C}(\text{BPA}_\delta)$,

$$\text{BPA}_\delta \vdash p = q \Leftrightarrow \boldsymbol{\mathcal{S}}(\text{BPA}_\delta) \models p = q.$$

Proof. (BAETEN AND VERHOEF, 1995) □

2.4.3 Process Algebra with inaction and renaming

One of the main goals of this thesis is to study formalisms for describing concurrent systems. However, the basic process-algebraic theories of the previous two subsections are only suitable for reasoning about sequential processes. To allow reasoning about concurrent processes, in this subsection, a concurrency operator is introduced. At the same time, a second new feature is added to the theory, namely renaming. Renaming is a general construct which can be used to define operators such as an encapsulation operator and an abstraction operator. The encapsulation operator renames actions to δ, thus effectively blocking, or *encapsulating*, them. The encapsulation operator is introduced below. The abstraction operator renames actions to τ, thus *hiding* them from the environment. The abstraction operator is introduced in the next subsection, where a process algebra for processes with silent behavior is given. Extending BPA$_\delta$ with concurrency and renaming yields the theory PA$_\delta$+RN, which is an abbreviation of Process Algebra with inaction and renaming. The absence of the adjective "basic" in the name expresses that the theory contains a concurrency operator. The naming conventions for ACP-style process algebras are standard (see, for example, BAETEN AND VERHOEF, 1995; BAETEN AND WEIJLAND, 1990).

The equational theory PA$_\delta$+RN. Let $\boldsymbol{A}$ be again some set of action labels. A renaming function $\boldsymbol{f} : (\boldsymbol{A} \cup \{\delta\}) \to (\boldsymbol{A} \cup \{\delta\})$ is any function with the restriction that $\boldsymbol{f}(\delta) = \delta$. This restriction means that the inaction process can be the result of a renaming, but that it cannot be renamed itself. It is included in the domain of renaming functions only for the sake of convenience. The sort of all renaming functions is denoted $\boldsymbol{F}$.

Table 2.4.19 gives the theory PA$_\delta$+RN. The informal explanation of the operators and the axioms is given below, along with the binding precedences of the operators.

PA$_\delta$+RN($\boldsymbol{A}$)			
BPA$_\delta$, $\boldsymbol{F}$			
$\_ \parallel \_,\ \_ \mathbin{⫗} \_ : \boldsymbol{P} \times \boldsymbol{P} \to \boldsymbol{P}$		$\rho_{\_} : \boldsymbol{F} \to (\boldsymbol{P} \to \boldsymbol{P})$	
$a : \boldsymbol{A} \cup \{\delta\};\ \boldsymbol{f} : \boldsymbol{F};\ x, y, z : \boldsymbol{P};$			
$x \parallel y = x \mathbin{⫗} y + y \mathbin{⫗} x$	$M1$	$\rho_f(a) = \boldsymbol{f}(a)$	$RN1$
$a \mathbin{⫗} x = a \cdot x$	$M2$	$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$	$RN2$
$a \cdot x \mathbin{⫗} y = a \cdot (x \parallel y)$	$M3$	$\rho_f(x \cdot y) = \rho_f(x) \cdot \rho_f(y)$	$RN3$
$(x + y) \mathbin{⫗} z = x \mathbin{⫗} z + y \mathbin{⫗} z$	$M4$		

Table 2.4.19: The process algebra PA$_\delta$+RN.

The theory PA$_\delta$+RN is an extension of BPA$_\delta$. Three new operators are introduced. The parallel-composition operator $\parallel$, often called the *merge* operator, denotes the parallel execution of its operands. It is axiomatized using an auxiliary operator $\mathbin{⫗}$, called the *left* merge. The left merge has the same meaning as the merge except that the left process must perform the first action. The axiomatization of the merge in terms of the left merge means that the interleaving view of concurrency is chosen (see also Example 2.2.5 on Page 6). Finally, for any renaming function $\boldsymbol{f}$ in $\boldsymbol{F}$, the renaming operator ρ_f simply renames actions according to $\boldsymbol{f}$. The binding precedence of the operators of $\mathrm{PA}_\delta{+}\mathrm{RN}$ is as follows. The unary renaming operators bind stronger than binary operators. Sequential composition binds stronger than all other binary operators. Choice binds weaker than all other operators.

The operators and axioms introduced in this subsection can be used to define various other process algebras. Extending the theory BPA with all the operators and axioms of Table 2.4.19 on the preceding page yields the process algebra PA+RN. Omitting the renaming operators and the corresponding axioms from $\mathrm{PA}_\delta + \mathrm{RN}$ or $\mathrm{PA} + \mathrm{RN}$ yields the theories PA_δ and PA respectively. It is also possible to extend BPA or BPA_δ with only renaming.

The following property states that the new operators introduced in this subsection can be eliminated from closed terms yielding an equivalent basic term.

Property 2.4.20. (Elimination)

i) For any closed term $p \in \mathcal{C}(X)$, *where* X *is a process algebra in* $\{\mathrm{PA}, \mathrm{PA{+}RN}\}$, *there is a basic term* $t \in \mathcal{B}(\mathrm{BPA})$, *such that* $X \vdash p = t$.

ii) For any closed term $p \in \mathcal{C}(X)$, *where* X *is a process algebra in* $\{\mathrm{PA}_\delta, \mathrm{PA}_\delta{+}\mathrm{RN}\}$, *there is a basic term* $t \in \mathcal{B}(\mathrm{BPA}_\delta)$, *such that* $X \vdash p = t$.

Proof. The results for PA and PA_δ appear in (BAETEN AND VERHOEF, 1995). Using the techniques of (BAETEN AND VERHOEF, 1995), the proofs for PA+RN and $\mathrm{PA}_\delta{+}\mathrm{RN}$ are straightforward. □

The next result states that the new theories are proper extensions of one of the basic theories BPA or BPA_δ. It is not possible to derive any equalities between BPA or BPA_δ processes from the axioms of the new theories that are not also derivable from the original theories.

Property 2.4.21. (Conservative extension)

i) For any closed terms $p, q \in \mathcal{C}(\mathrm{BPA})$ *and process algebra* $X \in \{\mathrm{PA}, \mathrm{PA{+}RN}\}$,
$X \vdash p = q \Leftrightarrow \mathrm{BPA} \vdash p = q$.

ii) For any closed terms $p, q \in \mathcal{C}(\mathrm{BPA}_\delta)$ *and process algebra* $X \in \{\mathrm{PA}_\delta, \mathrm{PA}_\delta{+}\mathrm{RN}\}$,
$X \vdash p = q \Leftrightarrow \mathrm{BPA}_\delta \vdash p = q$.

Proof. The results for PA and PA_δ appear in (BAETEN AND VERHOEF, 1995). Using the techniques of (BAETEN AND VERHOEF, 1995), the proofs for $\mathrm{PA} + \mathrm{RN}$ and $\mathrm{PA}_\delta + \mathrm{RN}$ are straightforward. Note that the proof techniques for proving conservativity results in (BAETEN AND VERHOEF, 1995) depend on the operational semantics given for the various process algebras. However, this does not cause problems, because the semantics given below and in the previous subsections are identical to those given in (BAETEN AND VERHOEF, 1995). The reason for presenting the conservativity result before the operational semantics is defined, is that the result itself is independent of the semantics. Furthermore, there are also proof techniques that do not use any specific semantics (see BAETEN AND WEIJLAND, 1990). □

Renaming and merge. The theory $PA_\delta+RN$ as introduced in Table 2.4.19 on Page 40 does not contain any axioms describing the interaction between the renaming operator and the merge. A meaningful and useful axiom would be the distribution of renaming over merge. The reason that such an axiom is not included in the standard definition of $PA_\delta+RN$ is twofold. First, the elimination result given above means that, in theory, such an axiom is redundant. Second, adding both distribution of renaming over merge and communication in the style of ACP (see BAETEN AND VERHOEF, 1995; BAETEN AND WEIJLAND, 1990) leads to an inconsistent theory.

However, for the purpose of this thesis, only process algebras without communication are needed. Since distribution of renaming over the merge operator is often convenient in verifications, two new axioms, given in Table 2.4.22, are introduced.

$\boldsymbol{f} : \boldsymbol{F};\ x, y : \boldsymbol{P};$	
$\rho_f(x \parallel y) = \rho_f(x) \parallel \rho_f(y)$	*RNM*1
$\rho_f(x \mathbin{\lfloor\!\lfloor} y) = \rho_f(x) \mathbin{\lfloor\!\lfloor} \rho_f(y)$	*RNM*2

Table 2.4.22: Axioms for renaming, merge, and left merge.

Note that Axiom *RNM*1 is derivable from Axioms *M*1 and *RNM*2. Hence, it might be more appropriate to refer to *RNM*1 as "Theorem *RNM*1" or "Property *RNM*1." However, it is tedious to have to distinguish between *RNM*1 and the other axioms. Therefore, for the sake of convenience, it is referred to as "Axiom *RNM*1." Axiom *RNM*2 is not derivable from the axioms of $PA_\delta + RN$ as introduced in Table 2.4.19 on Page 40. However, by means of the inductive proof techniques introduced below, it is possible to show that it can be derived from the axioms of $PA_\delta + RN$ *for closed terms*. Since inductive proofs are usually restricted to process algebras satisfying the elimination property, which not all process algebras in this thesis do, in the remainder, it is assumed that Axioms *RNM*1 and *RNM*2 are contained in the theory PA+RN. As a result, they are part of all extensions of PA+RN such as, for example, $PA_\delta+RN$. Note that the elimination and conservative-extension results given above are also valid for PA+RN and $PA_\delta+RN$ including Axioms *RNM*1 and *RNM*2.

The encapsulation operator. As mentioned earlier, the renaming operator can be used to define the so-called *encapsulation* operator. The encapsulation operator makes it possible to block occurrences of certain actions in a given process. Blocking all occurrences of an action is done by renaming that action to δ. Let H be a set of actions from $\boldsymbol{A}$. Let $\boldsymbol{f}_H$ be a renaming function in $\boldsymbol{F}$, which is defined as follows: $\boldsymbol{f}_H(\delta) = \delta$; for any $a \in H$, $\boldsymbol{f}_H(a) = \delta$ and for any $a \in \boldsymbol{A}\backslash H, \boldsymbol{f}_H(a) = a$. That is, the function $\boldsymbol{f}_H$ renames all actions in H to δ and does not affect any other actions. Consequently, the resulting renaming operator ρ_{f_H} exactly blocks all the actions in H. Since the operator ρ_{f_H} plays an important role in many ACP-like process algebras, it has its own notation, ∂_H, and is usually referred to as the encapsulation operator. Table 2.4.23 lists the renaming-operator axioms for the encapsulation operator. Axiom $RN1$ is split into two conditional axioms $D1$ and $D2$.

$a : \boldsymbol{A} \cup \{\delta\};\ H : \mathcal{P}(\boldsymbol{A});\ x, y : \boldsymbol{P};$

$a \notin H \Rightarrow \partial_H(a) = a$	$D1$
$a \in H \Rightarrow \partial_H(a) = \delta$	$D2$
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	$D3$
$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$	$D4$
$\partial_H(x \parallel y) = \partial_H(x) \parallel \partial_H(y)$	$DM1$
$\partial_H(x \mathbin{⌊\!⌊} y) = \partial_H(x) \mathbin{⌊\!⌊} \partial_H(y)$	$DM2$

Table 2.4.23: The axioms for the encapsulation operator.

An operational semantics for PA_δ+RN. The operational semantics for PA_δ+RN contains no surprises. The processes are the closed PA_δ+RN terms plus the special process $\surd$. The transition relation $\_ \xrightarrow{\_} \_ \subseteq (\mathcal{C}(\mathrm{PA}_\delta\mathrm{+RN}) \cup \{\surd\}) \times \boldsymbol{A} \times (\mathcal{C}(\mathrm{PA}_\delta\mathrm{+RN}) \cup \{\surd\})$ is the smallest relation satisfying the derivation rules given in Tables 2.4.5 on Page 36 and 2.4.24. In Table 2.4.5, of course, p, p', q, and q' range over $\mathcal{C}(\mathrm{PA}_\delta\mathrm{+RN})$.

$a : \boldsymbol{A};\ \boldsymbol{f} : \boldsymbol{F};\ p, p', q, q' : \mathcal{C}(\mathrm{PA}_\delta\mathrm{+RN});$

$$\frac{p \xrightarrow{a} p'}{p \parallel q \xrightarrow{a} p' \parallel q} \qquad \frac{q \xrightarrow{a} q'}{p \parallel q \xrightarrow{a} p \parallel q'} \qquad \frac{p \xrightarrow{a} \surd}{p \parallel q \xrightarrow{a} q} \qquad \frac{q \xrightarrow{a} \surd}{p \parallel q \xrightarrow{a} p}$$

$$\frac{p \xrightarrow{a} p'}{p \mathbin{⌊\!⌊} q \xrightarrow{a} p' \parallel q} \qquad \frac{p \xrightarrow{a} \surd}{p \mathbin{⌊\!⌊} q \xrightarrow{a} q}$$

$$\frac{p \xrightarrow{a} p', \quad \boldsymbol{f}(a) \neq \delta}{\rho_f(p) \xrightarrow{f(a)} \rho_f(p')} \qquad \frac{p \xrightarrow{a} \surd, \quad \boldsymbol{f}(a) \neq \delta}{\rho_f(p) \xrightarrow{f(a)} \surd}$$

Table 2.4.24: The transition relation for merge, left merge, and renaming.

Definition 2.4.25. (Operational semantics) The operational semantics $\boldsymbol{\mathcal{S}}(\mathrm{PA}_\delta\mathrm{+RN})$ is the process space $(\mathcal{C}(\mathrm{PA}_\delta\mathrm{+RN}) \cup \{\surd\}, \boldsymbol{A}, \longrightarrow, \{\surd\})$ with strong bisimilarity as the equivalence relation.

Property 2.4.26. (Congruence) *Strong bisimilarity, $\sim$, is a congruence for the operators of* PA_δ+RN.

Proof. The property is a straightforward consequence of several results in (BAETEN AND VERHOEF, 1995). $\square$

Theorem 2.4.27. (Soundness and completeness) *For closed terms* $p, q \in \mathcal{C}(\mathrm{PA}_\delta{+}\mathrm{RN})$,
$\mathrm{PA}_\delta{+}\mathrm{RN} \vdash p = q \Leftrightarrow \boldsymbol{\mathcal{S}}(\mathrm{PA}_\delta{+}\mathrm{RN}) \models p = q$.

Proof. Using the techniques of (BAETEN AND VERHOEF, 1995), the proof is straightforward. $\square$

Example 2.4.28. Consider again Example 2.2.5 on Page 6. The process depicted in Figure 2.2.4(d) on Page 7 can be specified in PA_δ+RN as follows: $rcmd \cdot pmat \cdot (omat \,\|\, sprdy)$. In Example 2.4.7 on Page 37, a BPA term representing the same process has already been given. Since these two terms have the same semantics and $\mathrm{PA}_\delta{+}\mathrm{RN}$ is complete for the operational semantics $\boldsymbol{\mathcal{S}}(\mathrm{PA}_\delta{+}\mathrm{RN})$, it should be possible to derive the equality of the two terms from the axioms of $\mathrm{PA}_\delta{+}\mathrm{RN}$. Axioms $M1$ and $M2$ yield the desired result.

It is interesting to compare the algebraic specification given in this example to the P/T net given in Figure 2.3.3 on Page 18. There are two major differences between the two formalisms. P/T nets have a graphical representation and explicitly represent states. Process-algebraic terms have a purely symbolic representation and do not explicitly represent states.

Structural induction. A proof technique that is often used in process algebra is induction on the structure of basic terms. To illustrate this proof technique, this paragraph contains an example. In this thesis, many results in Chapter 4 are proved by means of structural induction. The example also shows why the elimination property is useful. Note that by definition the composition of two renaming functions yields another renaming function.

Property 2.4.29. *For any closed term* $p \in \mathcal{C}(\mathrm{PA}_\delta{+}\mathrm{RN})$ *and* $\boldsymbol{f}, \boldsymbol{g} \in \boldsymbol{F}$,
$\mathrm{PA}_\delta{+}\mathrm{RN} \vdash \rho_{f \circ g}(p) = \rho_f(\rho_g(p))$.

Proof. It follows from Property 2.4.20 on Page 41 that there is a basic BPA_δ term t such that $\mathrm{PA}_\delta{+}\mathrm{RN} \vdash p = t$. Hence, it suffices to show that $\rho_{f \circ g}(t) = \rho_f(\rho_g(t))$. Given this result, it follows that $\mathrm{PA}_\delta{+}\mathrm{RN} \vdash \rho_{f \circ g}(p) = \rho_{f \circ g}(t) = \rho_f(\rho_g(t)) = \rho_f(\rho_g(p))$. The proof is by induction on the structure of basic BPA_δ terms. The symbol $\equiv$ is used to denote syntactical identity of terms.

i) Assume $t \equiv \delta$ or $t \equiv a$, for some action $a \in \boldsymbol{A}$. It follows that
$\rho_{f \circ g}(t) \overset{RN1}{=} \boldsymbol{f} \circ \boldsymbol{g}(t) = \boldsymbol{f}(\boldsymbol{g}(t)) \overset{RN1}{=} \rho_f(\boldsymbol{g}(t)) \overset{RN1}{=} \rho_f(\rho_g(t))$.

ii) Assume $t \equiv a \cdot s$, for some action $a \in \boldsymbol{A}$ and basic term $s \in \mathcal{B}(\mathrm{BPA}_\delta)$. Then,
$\rho_{f \circ g}(a \cdot s) \overset{RN3}{=} \rho_{f \circ g}(a) \cdot \rho_{f \circ g}(s) \overset{Induction\,(2\times)}{=} \rho_f(\rho_g(a)) \cdot \rho_f(\rho_g(s)) \overset{RN3}{=}$
$\rho_f(\rho_g(a) \cdot \rho_g(s)) \overset{RN3}{=} \rho_f(\rho_g(a \cdot s))$.

iii) Assume $t \equiv u + v$, for some basic terms u and v in $\mathcal{B}(\mathrm{BPA}_\delta)$. Then,
$\rho_{f \circ g}(u + v) \overset{RN2}{=} \rho_{f \circ g}(u) + \rho_{f \circ g}(v) \overset{Induction\,(2\times)}{=} \rho_f(\rho_g(u)) + \rho_f(\rho_g(v)) \overset{RN2}{=}$
$\rho_f(\rho_g(u) + \rho_g(v)) \overset{RN2}{=} \rho_f(\rho_g(u + v))$.

$\square$

Standard concurrency and expansion. Table 2.4.30 introduces the so-called *axioms of standard concurrency* . These axioms state properties of the merge operator that are intuitively meaningful and often useful in verifications. The reason for not including them in the basic theory is that $ASC2$ is derivable from the axioms of PA_δ+RN (see Property 2.4.31 below), whereas the other two are derivable for closed terms (see Property 2.4.32 below). The numbering of the axioms in Table 2.4.30 is not continuous, because a theory with both concurrency and communication has several more axioms of standard concurrency (see BAETEN AND VERHOEF, 1995; BAETEN AND WEIJLAND, 1990). The axioms that are only meaningful in a context with communication are omitted.

$x, y, z : \boldsymbol{P}$;	
$x \parallel y = y \parallel x$	$ASC2$
$(x \mathbin{\lfloor\!\lfloor} y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} (y \parallel z)$	$ASC4$
$(x \parallel y) \parallel z = x \parallel (y \parallel z)$	$ASC6$

Table 2.4.30: The axioms of standard concurrency.

Property 2.4.31. $\mathrm{PA}_\delta+\mathrm{RN} \vdash ASC2$.

Proof. It follows immediately from Axioms $M1$ and $A1$. □

The proof of the following property shows another use of induction in process algebra, namely natural induction on the number of symbols in one or more terms. Just like structural induction, it is a technique that is often used in process algebra. It also shows again the usefulness of the elimination property.

Property 2.4.32. *For any* $p, q,$ *and* r *in* $\mathcal{C}(\mathrm{PA}_\delta+\mathrm{RN})$,
$\mathrm{PA}_\delta+\mathrm{RN} \vdash (p \mathbin{\lfloor\!\lfloor} q) \mathbin{\lfloor\!\lfloor} r = p \mathbin{\lfloor\!\lfloor} (q \parallel r), (p \parallel q) \parallel r = p \parallel (q \parallel r)$.

Proof. It follows from Property 2.4.20 (Elimination) on Page 41 that it suffices to show the property for basic BPA_δ terms. The proof is by (strong) natural induction on the number of symbols in the terms. For any closed term $p \in \mathcal{C}(\mathrm{PA}_\delta\mathrm{+RN})$, let $l(p)$ denote the number of symbols in p. Let n be some natural number at least three. The following two properties must be proven.

$$(\forall s, t, u \; : \; s, t, u \in \mathcal{B}(\mathrm{BPA}_\delta) : \quad l(s) + l(t) + l(u) = n \Rightarrow \mathrm{PA}_\delta+\mathrm{RN} \vdash (s \mathbin{\lfloor\!\lfloor} t) \mathbin{\lfloor\!\lfloor} u = s \mathbin{\lfloor\!\lfloor} (t \parallel u) \quad) \text{ and} \tag{2.4.33}$$

$$(\forall s, t, u \; : \; s, t, u \in \mathcal{B}(\mathrm{BPA}_\delta) : \quad l(s) + l(t) + l(u) = n \Rightarrow \mathrm{PA}_\delta+\mathrm{RN} \vdash (s \parallel t) \parallel u = s \parallel (t \parallel u) \quad). \tag{2.4.34}$$

The induction hypothesis (IH) is as follows:

$(\forall k : 3 \leq k < n$
$\quad : (\forall s, t, u :\ s, t, u \in \mathcal{B}(\text{BPA}_\delta)$
$\qquad :\ k = l(s) + l(t) + l(u) \Rightarrow$
$\qquad\quad \text{PA}_\delta\text{+RN} \vdash (s \mathbin{⌊\!⌊} t) \mathbin{⌊\!⌊} u = s \mathbin{⌊\!⌊} (t \parallel u), (s \parallel t) \parallel u = s \parallel (t \parallel u)$
$\quad)$
$).$

Let s, t, and u be basic terms in $\mathcal{B}(\text{BPA}_\delta)$ such that $l(s) + l(t) + l(u) = n$. The proof of Property 2.4.33 consists of a case analysis.

i) Assume $s \equiv \delta$ or $s \equiv a$, for some action $a \in \boldsymbol{A}$. It easily follows that $(s \mathbin{⌊\!⌊} t) \mathbin{⌊\!⌊} u \overset{M2}{=} (s \cdot t) \mathbin{⌊\!⌊} u \overset{M3}{=} s \cdot (t \parallel u) \overset{M2}{=} s \mathbin{⌊\!⌊} (t \parallel u)$.

ii) Assume $s \equiv a \cdot v$, for some action $a \in \boldsymbol{A}$ and basic term $v \in \mathcal{B}(\text{BPA}_\delta)$. Then, $(a \cdot v \mathbin{⌊\!⌊} t) \mathbin{⌊\!⌊} u \overset{M3}{=} a \cdot (v \parallel t) \mathbin{⌊\!⌊} u \overset{M3}{=} a \cdot ((v \parallel t) \parallel u) \overset{IH}{=} a \cdot (v \parallel (t \parallel u)) \overset{M3}{=} a \cdot v \mathbin{⌊\!⌊} (t \parallel u)$.

iii) Assume $s \equiv v + w$, for some basic terms v and w in $\mathcal{B}(\text{BPA}_\delta)$. Then, $((v + w) \mathbin{⌊\!⌊} t) \mathbin{⌊\!⌊} u \overset{M4(2\times)}{=} (v \mathbin{⌊\!⌊} t) \mathbin{⌊\!⌊} u + (w \mathbin{⌊\!⌊} t) \mathbin{⌊\!⌊} u \overset{IH(2\times)}{=} v \mathbin{⌊\!⌊} (t \parallel u) + w \mathbin{⌊\!⌊} (t \parallel u) \overset{M4}{=} (v + w) \mathbin{⌊\!⌊} (t \parallel u)$.

This completes the proof of Property 2.4.33. Since the proof does not depend on Property 2.4.34, Property 2.4.33 may be used to prove Property 2.4.34. The proof is as follows.

$$
\begin{array}{ll}
 & (s \parallel t) \parallel u \\
= & \quad \{ M1(2\times) \} \\
 & (s \mathbin{⌊\!⌊} t + t \mathbin{⌊\!⌊} s) \mathbin{⌊\!⌊} u + u \mathbin{⌊\!⌊} (s \parallel t) \\
= & \quad \{ M4, ASC2 \} \\
 & (s \mathbin{⌊\!⌊} t) \mathbin{⌊\!⌊} u + (t \mathbin{⌊\!⌊} s) \mathbin{⌊\!⌊} u + u \mathbin{⌊\!⌊} (t \parallel s) \\
= & \quad \{ \text{Property 2.4.33 } (3\times) \} \\
 & s \mathbin{⌊\!⌊} (t \parallel u) + t \mathbin{⌊\!⌊} (s \parallel u) + (u \mathbin{⌊\!⌊} t) \mathbin{⌊\!⌊} s \\
= & \quad \{ ASC2, \text{Property 2.4.33} \} \\
 & s \mathbin{⌊\!⌊} (t \parallel u) + (t \mathbin{⌊\!⌊} u) \mathbin{⌊\!⌊} s + (u \mathbin{⌊\!⌊} t) \mathbin{⌊\!⌊} s \\
= & \quad \{ M4, M1(2\times) \} \\
 & s \parallel (t \parallel u)
\end{array}
$$

The principle of strong natural induction implies that Properties 2.4.33 and 2.4.34 hold for all natural numbers at least three, which completes the proof of Property 2.4.32.

A slightly different proof of Property 2.4.32, which does not use natural induction but structural induction, can be found in (BAETEN AND WEIJLAND, 1990). □

For the interested reader, it is a nice exercise to prove by means of strong natural induction that, for closed PA_δ+RN terms, Axiom *RNM*2 in Table 2.4.22 on Page 42 is derivable from the other axioms of theory PA_δ+RN given in Table 2.4.19 on Page 40.

It follows immediately from Properties 2.4.31 and 2.4.32 on the preceding page that the axioms of standard concurrency are valid in the operational semantics $\mathcal{S}(\text{PA}_\delta\text{+RN})$ given in Definition 2.4.25 on Page 43.

Property 2.4.35. *For any* $p, q, r \in \mathcal{C}(\text{PA}_\delta\text{+RN})$,
$\mathcal{S}(\text{PA}_\delta\text{+RN}) \models p \parallel q = q \parallel p, (p \mathbin{⌊\!⌊} q) \mathbin{⌊\!⌊} r = p \mathbin{⌊\!⌊} (q \parallel r), (p \parallel q) \parallel r = p \parallel (q \parallel r)$.

Proof. Properties 2.4.31 and 2.4.32 and Theorem 2.4.27 (Soundness) on Page 44. □

In PA_δ+RN extended with the axioms of standard concurrency, it is possible to prove the following theorem, which is very useful for simplifying many calculations. It states that a parallel composition of k processes, where k is some natural number at least two, can be expanded into a choice of exactly those k alternatives where in each of the alternatives a different one of the k processes performs the initial action. For any *associative* binary process-algebra operator $\otimes$, some function f, some $n \in \mathbb{P}$, and operands $x_0, \ldots, x_n$, the quantifier notation $(\otimes\, i : 0 \leq i \leq n : f(x_i))$ is used as a shorthand notation for $f(x_0) \otimes \ldots \otimes f(x_n)$.

Theorem 2.4.36. (Expansion) *Let* $x_0, \ldots, x_n$ *be terms of sort* $\boldsymbol{P}$*, where* $n \in \mathbb{P}$*.*
PA_δ+RN+standard concurrency $\vdash$

$$(\|\, i : 0 \leq i \leq n : x_i) = (+\, i : 0 \leq i \leq n : x_i \mathbin{\lfloor\!\lfloor} (\|\, j : 0 \leq j \leq n \wedge j \neq i : x_j)).$$

Proof. The proof by induction on n is taken from (BAETEN AND WEIJLAND, 1990). The base case, $n = 1$, is Axiom $M1$. Assume $n > 1$.

$$\begin{array}{ll}
 & (\|\, i : 0 \leq i \leq n : x_i) \\
= & \quad \{\, ASC6 \text{ (associativity of merge) } \} \\
 & (\|\, i : 0 \leq i < n : x_i) \parallel x_n \\
= & \quad \{\, M1 \,\} \\
 & (\|\, i : 0 \leq i < n : x_i) \mathbin{\lfloor\!\lfloor} x_n + x_n \mathbin{\lfloor\!\lfloor} (\|\, i : 0 \leq i < n : x_i) \\
= & \quad \{\, \text{Induction} \,\} \\
 & (+\, i : 0 \leq i < n : x_i \mathbin{\lfloor\!\lfloor} (\|\, j : 0 \leq j < n \wedge j \neq i : x_j)) \mathbin{\lfloor\!\lfloor} x_n + \\
 & \quad x_n \mathbin{\lfloor\!\lfloor} (\|\, i : 0 \leq i < n : x_i) \\
= & \quad \{\, M4, ASC4 \,\} \\
 & (+\, i : 0 \leq i < n : x_i \mathbin{\lfloor\!\lfloor} ((\|\, j : 0 \leq j < n \wedge j \neq i : x_j) \parallel x_n)) + \\
 & \quad x_n \mathbin{\lfloor\!\lfloor} (\|\, i : 0 \leq i < n : x_i) \\
= & \quad \{\, ASC6 \text{ (associativity of merge); dummy change; calculus } \} \\
 & (+\, i : 0 \leq i < n : x_i \mathbin{\lfloor\!\lfloor} (\|\, j : 0 \leq j \leq n \wedge j \neq i : x_j)) + \\
 & \quad x_n \mathbin{\lfloor\!\lfloor} (\|\, j : 0 \leq j \leq n \wedge j \neq n : x_j) \\
= & \quad \{\, A2 \text{ (associativity of choice) } \} \\
 & (+\, i : 0 \leq i \leq n : x_i \mathbin{\lfloor\!\lfloor} (\|\, j : 0 \leq j \leq n \wedge j \neq i : x_j))
\end{array}$$
□

2.4.4 Process Algebra with inaction, silent actions, and renaming

So far in this section, no distinction has been made between observable behavior and silent behavior. This subsection extends the process algebra PA_δ+RN in such a way that it is possible to reason about processes with silent behavior.

The equational theory PA_δ^τ+RN. The introduction of the silent action τ in the process-algebraic setting developed so far, yields the process algebra PA_δ^τ+RN. The acronym PA_δ^τ+RN is an abbreviation of Process Algebra with inaction, silent actions, and renaming.

$\_\mathrm{PA}_\delta^\tau + \mathrm{RN}(\boldsymbol{A})$	
$\mathrm{PA}_\delta + \mathrm{RN}(\boldsymbol{A})$	
$\tau : \boldsymbol{A}$	
$x, y, z : \boldsymbol{P};$	
$x \cdot \tau = x$	$B1$
$x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z)$	$B2$

Table 2.4.37: The process algebra PA_δ^τ+RN.

Let $\boldsymbol{A}$ be some set of action labels. Table 2.4.37 gives the theory PA_δ^τ+RN. Note that the second entry of Table 2.4.37 introduces the special action τ.

Axioms $B1$ and $B2$ are the basis for an axiomatization of rooted branching bisimilarity (see VAN GLABBEEK AND WEIJLAND, 1996). These axioms are, of course, chosen because rooted branching bisimilarity is the equivalence used in the operational semantics given below. Together, $B1$ and $B2$ state that it is allowed to remove a silent action provided that it does not enforce an internal choice.

Example 2.4.38. Consider two versions of a production unit which are specified as follows: $rcmd \cdot (\tau \cdot (pmat_1 + pmat_2)) \cdot omat$ and $rcmd \cdot (\tau \cdot pmat_1 + pmat_2) \cdot omat$. The first production unit decides after some internal computations what source of input material to use. Clearly, the environment cannot distinguish it from a unit with behavior $rcmd \cdot (pmat_1 + pmat_2) \cdot omat$. In PA_δ^τ+RN, the equality follows simply from Axioms $A5$ and $B1$. The second production unit behaves slightly different. Based on some internal computations, it may decide autonomously that it wants to get input from source one. Thus, it removes the option to get input from the other source. The environment can detect this internal choice when input source one is left empty. When making enough runs with the unit, it may get stuck in a deadlock once in a while. The algebra PA_δ^τ+RN does not contain any axioms to remove the silent action in the unit's specification.

In the previous subsection, renaming functions have been introduced. In the presence of silent actions, renaming functions must satisfy an additional requirement. Since silent actions are internal and unobservable, they cannot be renamed. That is, for any $\boldsymbol{f}$ in the sort of renaming functions $\boldsymbol{F}, \boldsymbol{f}(\tau)$ must be τ. One of the consequences is that silent actions cannot be blocked. This implies that the definition of the encapsulation operator when defined in PA_δ^τ+RN must be adapted accordingly. The axioms in Table 2.4.23 on Page 43 remain the same, but H now ranges over $\mathcal{P}(\boldsymbol{A} \backslash \{\tau\})$.

Silent actions can be used to define a whole range of new process algebras. The most interesting ones are PA^τ, PA^τ+RN, and PA_δ^τ. By now, the exact definition of these theories should need no further elaboration. The following property states several elimination results.

Property 2.4.39. (Elimination)

i) *For any closed term $p \in \mathcal{C}(X)$, where X is a process algebra in* $\{\mathrm{PA}^\tau, \mathrm{PA}^\tau + \mathrm{RN}\}$, *there is a basic term $t \in \mathcal{B}(\mathrm{BPA})$, such that $X \vdash p = t$.*

ii) For any closed term $p \in \mathcal{C}(X)$, where X is a process algebra in $\{\mathrm{PA}_\delta^\tau, \mathrm{PA}_\delta^\tau+\mathrm{RN}\}$, there is a basic term $t \in \mathcal{B}(\mathrm{BPA}_\delta)$, such that $X \vdash p = t$.

Proof. The proof is identical to the proof of Property 2.4.20 on Page 41. The introduction of the silent action τ has no effect. □

In addition to the above elimination results, various conservativity results can be proved. It goes beyond the scope of this chapter to present all these results. Basically, extending a theory with any of the operators given so far yields a conservative extension.

The abstraction operator. The process algebra PA_δ^τ+RN allows us to specify processes with internal behavior. However, does it also provide an abstraction facility? That is, is it possible to *hide* certain behavior of a process from the environment? The answer is affirmative. Hiding the occurrence of an action is established by renaming that action occurrence to τ. This means that the renaming operator can be used to define an abstraction operator. The definition is very similar to the definition of the encapsulation operator in the previous subsection. Let I be a set of actions from $\boldsymbol{A}\backslash\{\tau\}$. Let $\boldsymbol{f}_I$ be a renaming function in $\boldsymbol{F}$, which is defined as follows: $\boldsymbol{f}_I(\delta) = \delta$; for any $a \in I$, $\boldsymbol{f}_I(a) = \tau$ and for any $a \in \boldsymbol{A}\backslash I$, $\boldsymbol{f}_I(a) = a$. Note that this definition satisfies the requirement that $\boldsymbol{f}_I(\tau) = \tau$. Function $\boldsymbol{f}_I$ renames exactly all actions in I to τ. The resulting renaming operator ρ_{f_I} thus effectively hides the actions in I from the environment. Because of its importance in process algebra, the operator ρ_{f_I} is usually denoted τ_I, and is referred to as the abstraction operator. Table 2.4.40 lists the renaming-operator axioms for the abstraction operator.

$a : \boldsymbol{A} \cup \{\delta\};\ I : \mathcal{P}(\boldsymbol{A}\backslash\{\tau\});\ x, y : \boldsymbol{P};$	
$a \notin I \Rightarrow \tau_I(a) = a$	*TI*1
$a \in I \Rightarrow \tau_I(a) = \tau$	*TI*2
$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	*TI*3
$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$	*TI*4
$\tau_I(x \parallel y) = \tau_I(x) \parallel \tau_I(y)$	*TIM*1
$\tau_I(x \mathbin{\lfloor\!\lfloor} y) = \tau_I(x) \mathbin{\lfloor\!\lfloor} \tau_I(y)$	*TIM*2

Table 2.4.40: The axioms for the abstraction operator.

An operational semantics for PA_δ^τ+RN. The processes in the operational semantics for PA_δ^τ+RN are the closed PA_δ^τ+RN terms extended with the special process $\surd$. The transition relation $\_ \xrightarrow{\_} \_ \subseteq (\mathcal{C}(\mathrm{PA}_\delta^\tau\text{+RN}) \cup \{\surd\}) \times \boldsymbol{A} \times (\mathcal{C}(\mathrm{PA}_\delta^\tau\text{+RN}) \cup \{\surd\})$ is identical to the transition relation for PA_δ+RN. That is, it is the smallest relation satisfying the derivation rules given in Tables 2.4.5 on Page 36 and 2.4.24 on Page 43, where in both tables p, p', q, and q' range over $\mathcal{C}(\mathrm{PA}_\delta^\tau\text{+RN})$. The only difference with the operational semantics for PA_δ+RN is that rooted branching bisimilarity is chosen as the equivalence relation.

Definition 2.4.41. (Operational semantics) The operational semantics $\mathcal{S}(\mathrm{PA}_\delta^\tau+\mathrm{RN})$ is the process space $(\mathcal{C}(\mathrm{PA}_\delta^\tau+\mathrm{RN}) \cup \{\surd\}, \boldsymbol{A}, \longrightarrow, \{\surd\})$ with rooted branching bisimilarity as the equivalence relation.

As mentioned in Section 2.2.3, the root condition in the definition of rooted branching bisimilarity guarantees that the equivalence is a congruence for all the algebraic operators.

Property 2.4.42. (Congruence) *Rooted branching bisimilarity, $\sim_{rb}$, is a congruence for the operators of* $\mathrm{PA}_\delta^\tau+\mathrm{RN}$.

Proof. Let p_1, p_2, q_1, q_2 be closed $\mathrm{PA}_\delta^\tau+\mathrm{RN}$ terms. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be rooted branching bisimulations between p_1 and q_1, and p_2 and q_2 respectively. It must be shown that there exist rooted branching bisimulations $\mathcal{Q}_1, \ldots, \mathcal{Q}_5$, such that $(p_1+p_2)\mathcal{Q}_1(q_1+q_2)$, $(p_1 \cdot p_2)\mathcal{Q}_2(q_1 \cdot q_2)$, $(p_1 \parallel p_2)\mathcal{Q}_3(q_1 \parallel q_2)$, $(p_1 \mathbin{⌊\!⌊} p_2)\mathcal{Q}_4(q_1 \mathbin{⌊\!⌊} q_2)$, and, such that, for any renaming function $\boldsymbol{f} \in \boldsymbol{F}$, $\rho_f(p_1)\mathcal{Q}_5\rho_f(q_1)$. The five relations are given below. It is straightforward to verify that the relations are indeed rooted branching bisimulations. To avoid unnecessarily complex formulas, some notations are introduced. Let, for any $p \in \mathcal{C}(\mathrm{PA}_\delta^\tau+\mathrm{RN}) \cup \{\surd\}$, $\surd \cdot p = p$, $\surd \parallel p = p \parallel \surd = p$, and, for any $\boldsymbol{f} \in \boldsymbol{F}$, $\rho_f(\surd) = \surd$.

$\mathcal{Q}_1 = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{(p_1+p_2, q_1+q_2)\}$,
$\mathcal{Q}_2 = \mathcal{R}_2 \cup \{(p \cdot p_2, q \cdot q_2) \mid p\mathcal{R}_1 q\}$,
$\mathcal{Q}_3 = \{(p \parallel p', q \parallel q') \mid p\mathcal{R}_1 q \wedge p'\mathcal{R}_2 q'\}$,
$\mathcal{Q}_4 = \mathcal{Q}_3 \cup \{(p_1 \mathbin{⌊\!⌊} p_2, q_1 \mathbin{⌊\!⌊} q_2)\}$,
$\mathcal{Q}_5 = \{(\rho_f(p), \rho_f(q)) \mid p\mathcal{R}_1 q \wedge \boldsymbol{f} \in \boldsymbol{F}\}$.

□

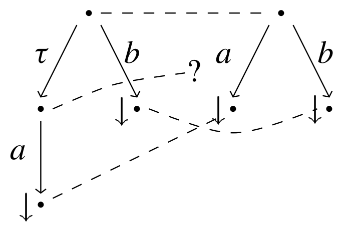


Figure 2.4.43: Branching bisimilarity is not a congruence for the choice operator.

Example 2.4.44. The reason for introducing the root condition in Section 2.2.3 was that branching bisimilarity is not a congruence for the algebraic choice operator, as the following example shows. Let a and b be two different actions in $\boldsymbol{A}$. Processes $\tau \cdot a$ and a are branching bisimilar. Figure 2.2.17 on Page 12 shows a branching bisimulation between these two processes. Clearly, also process b is branching bisimilar to itself. However, $\tau \cdot a + b$ is not branching bisimilar to $a + b$. Figure 2.4.43 shows the problem when trying to construct a branching bisimulation. Process $\tau \cdot a + b$ has a state in which only a is possible, whereas $a + b$ does not have such a state. The problem does not occur for rooted branching bisimilarity, because $\tau \cdot a$ and a are not rooted branching bisimilar (see also Example 2.2.18 on Page 12).

Theorem 2.4.45. (Soundness and completeness) *For closed terms* $p, q \in \mathcal{C}(\mathrm{PA}_\delta^\tau{+}\mathrm{RN})$,
$\mathrm{PA}_\delta^\tau{+}\mathrm{RN} \vdash p = q \Leftrightarrow \mathcal{S}(\mathrm{PA}_\delta^\tau{+}\mathrm{RN}) \models p = q$.

Proof. The theorem is a fairly straightforward result of Property 2.4.42, the completeness of BPA^τ for rooted branching bisimilarity (see VAN GLABBEEK AND WEIJLAND, 1996) and the proof techniques in (BAETEN AND VERHOEF, 1995). □

Example 2.4.46. Consider Examples 2.2.12 on Page 10 and 2.2.20 on Page 13. The process in Figure 2.2.11(b) on Page 9 can be specified in $\mathrm{PA}_\delta^\tau{+}\mathrm{RN}$ as follows: $rcmd \cdot (pmat_1 + pmat_2) \cdot omat$ (see also Example 2.4.8 on Page 37). In Example 2.2.20, the claim has been made that the process of Figure 2.2.19(a) on Page 13 is the result of abstracting away the processing actions $pmat_1$ and $pmat_2$ in the process of Figure 2.2.11(b). The process of Figure 2.2.19(a) is specified by the $\mathrm{PA}_\delta^\tau{+}\mathrm{RN}$ term $rcmd \cdot \tau \cdot omat$. In Example 2.2.20, it has also been shown that the resulting observable behavior is the process in Figure 2.2.19(b), which is specified in $\mathrm{PA}_\delta^\tau{+}\mathrm{RN}$ as $rcmd \cdot omat$. Using the abstraction operator of $\mathrm{PA}_\delta^\tau{+}\mathrm{RN}$, the abstraction of actions can be formalized and it is possible to calculate the resulting observable behavior. Let I be equal to $\{pmat_1, pmat_2\}$. The following derivation shows the desired results.

$$
\begin{array}{ll}
 & \tau_I(rcmd \cdot (pmat_1 + pmat_2) \cdot omat) \\
= & \quad \{ \ TI3, TI4 \text{ (repeatedly) } \} \\
 & \tau_I(rcmd) \cdot (\tau_I(pmat_1) + \tau_I(pmat_2)) \cdot \tau_I(omat) \\
= & \quad \{ \ TI1, TI2 \text{ (repeatedly) } \} \\
 & rcmd \cdot (\tau + \tau) \cdot omat \\
= & \quad \{ \ A3 \ \} \\
 & rcmd \cdot \tau \cdot omat \\
= & \quad \{ \ B1 \ \} \\
 & rcmd \cdot omat
\end{array}
$$

Structural induction, standard concurrency, and expansion. Since $\mathrm{PA}_\delta^\tau{+}\mathrm{RN}$ has an elimination result (Property 2.4.39 on Page 48), structural induction in $\mathrm{PA}_\delta^\tau + \mathrm{RN}$ goes in exactly the same way as in Section 2.4.3, where an example in $\mathrm{PA}_\delta{+}\mathrm{RN}$ is given. The extension of the theory $\mathrm{PA}_\delta^\tau{+}\mathrm{RN}$ with the axioms of standard concurrency is also identical to the extension of $\mathrm{PA}_\delta + \mathrm{RN}$ with standard concurrency. As a result, $\mathrm{PA}_\delta^\tau + \mathrm{RN}$ has the same expansion theorem as the one given in Theorem 2.4.36 on Page 47.

2.4.5 Iteration in process algebra

This subsection introduces a simple form of recursion into process algebra, namely *iteration*, in the form of the *binary Kleene star*. The binary Kleene star is the original star operator as introduced in (KLEENE, 1956). The binary Kleene star is much simpler than general recursion (see BAETEN AND VERHOEF, 1995; BAETEN AND WEIJLAND, 1990).

The equational theory $\mathrm{PA}_\delta^{\tau *}+\mathrm{RN}$. Table 2.4.47 gives the theory $\mathrm{PA}_\delta^{\tau *}+\mathrm{RN}$, Process Algebra with inaction, silent actions, renaming, and iteration.

$\mathrm{PA}_\delta^{\tau *}+\mathrm{RN}(\boldsymbol{A})$	
$\mathrm{PA}_\delta^{\tau}+\mathrm{RN}(\boldsymbol{A})$	
$\_{}^*\_ : \boldsymbol{P} \times \boldsymbol{P} \to \boldsymbol{P}$	
$\boldsymbol{f} : \boldsymbol{F};\ x, y, z : \boldsymbol{P};$	
$x^* y = x \cdot (x^* y) + y$	*BKS*1
$x^*(y \cdot z) = (x^* y) \cdot z$	*BKS*2
$x^*(y \cdot ((x+y)^* z) + z) = (x+y)^* z$	*BKS*3
$\rho_f(x^* y) = \rho_f(x)^* \rho_f(y)$	*BKS*4

Table 2.4.47: The process algebra $\mathrm{PA}_\delta^{\tau *}+\mathrm{RN}$.

The binary Kleene star * was introduced in process algebra in (BERGSTRA, BETHKE, AND PONSE, 1994), where also the axioms *BKS*1 through *BKS*4 appear. ((BERGSTRA, BETHKE, AND PONSE, 1994) gives variants of *BKS*4 for the encapsulation and abstraction operators.) Axiom *BKS*3 is a sophisticated axiom which is needed to get a complete axiomatization of bisimilarity. Below, completeness is discussed in more detail. It is possible to extend any of the other theories given so far with the binary Kleene star. In case the basic theory does not contain the renaming operator, *BKS*4 is omitted. The binary Kleene star has the same binding priority as sequential composition.

A disadvantage of iteration is that elimination of the binary Kleene star from closed terms is not possible anymore (see BAETEN AND VERHOEF, 1995). As a result, inductive proofs as given in the Section 2.4.3 do not carry over to a theory containing the binary Kleene star. One consequence is that the axioms of standard concurrency *ASC*4 and *ASC*6 are no longer derivable for closed terms (see Property 2.4.32 on Page 45). Note that they can still be *valid* in some given operational semantics (see Property 2.4.35 on Page 46).

An operational semantics for $\mathrm{PA}_\delta^{\tau *}+\mathrm{RN}$. The processes in the operational semantics for $\mathrm{PA}_\delta^{\tau *}+\mathrm{RN}$ are the closed $\mathrm{PA}_\delta^{\tau *}+\mathrm{RN}$ terms extended with the special process $\surd$. The transition relation $\_ \xrightarrow{-} \_ \subseteq (\mathcal{C}(\mathrm{PA}_\delta^{\tau *}+\mathrm{RN}) \cup \{\surd\}) \times \boldsymbol{A} \times (\mathcal{C}(\mathrm{PA}_\delta^{\tau *}+\mathrm{RN}) \cup \{\surd\})$ is the smallest relation satisfying the derivation rules given in Tables 2.4.5 on Page 36, 2.4.24 on Page 43, and 2.4.48 on the next page, where in all tables p, p', q, and q' range over $\mathcal{C}(\mathrm{PA}_\delta^{\tau *}+\mathrm{RN})$.

Definition 2.4.49. (Operational semantics) The operational semantics $\boldsymbol{\mathcal{S}}(\mathrm{PA}_\delta^{\tau *}+\mathrm{RN})$ is the process space $(\mathcal{C}(\mathrm{PA}_\delta^{\tau *}+\mathrm{RN}) \cup \{\surd\}, \boldsymbol{A}, \longrightarrow, \{\surd\})$ with rooted branching bisimilarity as the equivalence relation.

Example 2.4.50. In theory $\mathrm{PA}_\delta^{\tau *}+\mathrm{RN}$, it is finally possible to specify the process depicted in Figure 2.2.4 (c) on Page 7: $(rcmd \cdot pmat \cdot omat)^* \delta$. The non-terminating iteration is specified using the binary Kleene star in combination with the inaction constant δ.

$a : \boldsymbol{A};\ p, p', q, q' : \mathcal{C}(\mathrm{PA}_\delta^{\tau *}\text{+RN});$

$$\frac{p \xrightarrow{a} p'}{p^* q \xrightarrow{a} p' \cdot (p^* q)} \qquad \frac{q \xrightarrow{a} q'}{p^* q \xrightarrow{a} q'} \qquad \frac{p \xrightarrow{a} \surd}{p^* q \xrightarrow{a} p^* q} \qquad \frac{q \xrightarrow{a} \surd}{p^* q \xrightarrow{a} \surd}$$

Table 2.4.48: The transition relation for the binary Kleene star.

Property 2.4.51. (Congruence) *Rooted branching bisimilarity, $\sim_{rb}$, is a congruence for the operators of* $\mathrm{PA}_\delta^{\tau *}$+RN.

Proof. For all operators of $\mathrm{PA}_\delta^{\tau *}$+RN except the binary Kleene star, the proof is identical to the proof of Property 2.4.42 on Page 50. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be rooted branching bisimulations between p_1 and q_1, and p_2 and q_2, respectively, where p_1, p_2, q_1, and q_2 are closed $\mathrm{PA}_\delta^{\tau *}$+ RN terms. Let, for any $p \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}\text{+RN}) \cup \{\surd\}$, $\surd \cdot p$ denote p. It is straightforward to verify that $\mathcal{R}_2 \cup \{(p_1 {}^* p_2, q_1 {}^* q_2)\} \cup \{(p \cdot (p_1 {}^* p_2), q \cdot (q_1 {}^* q_2)) \mid p \mathcal{R}_1 q\}$ is a rooted branching bisimulation between $p_1 {}^* p_2$ and $q_1 {}^* q_2$. □

Theorem 2.4.52. (Soundness) *For any closed terms $p, q \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}\text{+RN})$,*
$\mathrm{PA}_\delta^{\tau *}\text{+RN} \vdash p = q \Rightarrow \mathcal{S}(\mathrm{PA}_\delta^{\tau *}\text{+RN}) \models p = q.$

Proof. Since rooted branching bisimilarity is a congruence for the operators of $\mathrm{PA}_\delta^{\tau *}$+RN, it suffices to show the validity of each of the axioms. It is not difficult to construct a rooted branching bisimulation for each case. □

There is no completeness result for $\mathrm{PA}_\delta^{\tau *}$+RN. On the contrary, the results of (SEWELL, 1994) show that in a setting with the binary Kleene star and inaction, no finite complete axiomatization for strong bisimilarity exists. The techniques of (SEWELL, 1994) can also be used to prove that in a context with the binary Kleene star and silent actions no finite complete axiomatization exists for (rooted) branching bisimilarity. The only completeness result for the binary Kleene star that to date exists is a result proved in (FOKKINK AND ZANTEMA, 1994). There, it is shown that BPA*, Basic Process Algebra with the binary Kleene star, is a sound and complete axiomatization for strong bisimilarity on closed terms. In addition, there are several completeness results for restricted forms of the binary Kleene star (see VAN GLABBEEK, 1997, for an overview).

Standard concurrency and expansion. Although the axioms of standard concurrency *ASC*4 and *ASC*6 are not derivable from the axioms of $\mathrm{PA}_\delta^{\tau *}$+RN, they are valid in the semantics $\mathcal{S}(\mathrm{PA}_\delta^{\tau *}\text{+RN})$. Axiom *ASC*2 is still derivable from $\mathrm{PA}_\delta^{\tau *}$+RN, and hence also valid in $\mathcal{S}(\mathrm{PA}_\delta^{\tau *}\text{+RN})$.

Property 2.4.53. *For any $p, q, r \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}\text{+RN})$,*
$\mathcal{S}(\mathrm{PA}_\delta^{\tau *}\text{+RN}) \models p \parallel q = q \parallel p, (p \mathbin{⌊\!⌊} q) \mathbin{⌊\!⌊} r = p \mathbin{⌊\!⌊} (q \parallel r), (p \parallel q) \parallel r = p \parallel (q \parallel r).$

Proof. Property 2.4.31 on Page 45, adapted for $\mathrm{PA}_\delta^{\tau *}$+RN, and Theorem 2.4.52 (Soundness) immediately yield that $\mathcal{S}(\mathrm{PA}_\delta^{\tau *}\text{+RN}) \models p \parallel q = q \parallel p$. The following relation is a rooted branching bisimulation both between $(p \mathbin{⌊\!⌊} q) \mathbin{⌊\!⌊} r$ and $p \mathbin{⌊\!⌊} (q \parallel r)$ and between $(p \parallel q) \parallel r$ and $p \parallel (q \parallel r)$:

$$\{((p \mathbin{⌊\!⌊} q) \mathbin{⌊\!⌊} r, p \mathbin{⌊\!⌊} (q \parallel r)) \mid p, q, r \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}{+}\mathrm{RN})\}$$
$$\cup \{((p \parallel q) \parallel r, p \parallel (q \parallel r)) \mid p, q, r \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}{+}\mathrm{RN})\}$$
$$\cup \{(p, p) \mid p \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}{+}\mathrm{RN}) \cup \{\surd\}\}.$$

□

A consequence of Property 2.4.53 is that $\mathrm{PA}_\delta^{\tau *}$+RN+standard concurrency has the same expansion theorem as given in Theorem 2.4.36 on Page 47. The addition of the binary Kleene star does not influence the proof of this theorem.

Fairness. In a theory which includes both recursion and silent actions, it is often useful to have a *fairness* principle. A simple form of fairness means that a process cannot always choose to do an internal action if it also has other options. For the binary Kleene star, this form of fairness can be expressed by a single axiom, namely the *Fair Iteration Rule* (*FIR*) given in Table 2.4.54. It simply states that a sequence of silent actions cannot be infinitely long. For a treatment of other, more sophisticated forms of fairness in process algebra, the reader is referred to (BAETEN AND WEIJLAND, 1990).

x : ***P***;	
$\tau * x = x + \tau \cdot x$	*FIR*

Table 2.4.54: The Fair Iteration Rule.

The Fair Iteration Rule is valid in the operational semantics $\mathcal{S}(\mathrm{PA}_\delta^{\tau *}{+}\mathrm{RN})$ of Definition 2.4.49 on Page 52.

Property 2.4.55. *For any* $p \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}{+}\mathrm{RN})$,
$\mathcal{S}(\mathrm{PA}_\delta^{\tau *}{+}\mathrm{RN}) \models \tau * p = p + \tau \cdot p.$

Proof. For any $p \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}{+}\mathrm{RN})$, the following relation is a rooted branching bisimulation between $\tau * p$ and $p + \tau \cdot p$.

$$\{(\tau * p, p + \tau \cdot p) \mid p \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}{+}\mathrm{RN})\}$$
$$\cup \{(\tau * p, p) \mid p \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}{+}\mathrm{RN})\} \cup \{(p, p) \mid p \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}{+}\mathrm{RN}) \cup \{\surd\}\}.$$

□

Recursive equations. Since often verifications in process algebra result in an equation that is recursive in some given process, it is useful to have a principle that gives solutions for such recursive equations. The *Recursive Specification Principle* for the binary Kleene star (*RSP**) given in Table 2.4.57 on the next page is a derivation rule which gives a solution in terms of the binary Kleene star for some restricted set of recursive equations. Axiom *RSP** uses the notion of a guard which is defined as follows. Informally, a term is a guard if it cannot terminate successfully without performing at least one observable action.

Definition 2.4.56. (Guard) A *closed* $\mathrm{PA}_\delta^{\tau *}$+RN term p is a *guard* if and only if, using the axioms of $\mathrm{PA}_\delta^{\tau *}$+RN, it can be rewritten into an equivalent term of any of the following forms:

i) δ or a, for any $a \in \boldsymbol{A} \backslash \{\tau\}$;

ii) $q \cdot r$, for closed $\mathrm{PA}_\delta^{\tau *}$+RN terms q and r where either q or r or both are guards;

iii) $q + r$, for any closed $\mathrm{PA}_\delta^{\tau *}$+RN terms q and r where both q and r are guards.

$x, y, z : \boldsymbol{P}$;	
$\dfrac{x = y \cdot x + z, \quad y \text{ is a guard}}{x = y * z}$	*RSP**

Table 2.4.57: The Recursive Specification Principle for the binary Kleene star.

Example 2.4.58. Definition 2.4.56 states that silent actions cannot act as guards. The following example shows the reason for this restriction. Let a and b be two different actions in $\boldsymbol{A}$ not equal to τ. Axioms $A5$, $A6$, and $B1$ easily yield that $\tau \cdot a = \tau \cdot (\tau \cdot a) + \delta$ and $\tau \cdot b = \tau \cdot (\tau \cdot b) + \delta$. Assume that the silent action is a guard. Then, it follows from *RSP** that $\tau \cdot a = \tau * \delta$ and $\tau \cdot b = \tau * \delta$. Hence, $\tau \cdot a = \tau \cdot b$, which is clearly a contradiction when a and b are different.

The derivation rule *RSP** is valid in the operational semantics $\boldsymbol{\mathcal{S}}(\mathrm{PA}_\delta^{\tau *}\text{+RN})$.

Property 2.4.59. *For any* $p, q, r \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}\text{+RN})$ *such that* q *is a guard,*
$\boldsymbol{\mathcal{S}}(\mathrm{PA}_\delta^{\tau *}\text{+RN}) \models p = q \cdot p + r \;\Rightarrow\; \boldsymbol{\mathcal{S}}(\mathrm{PA}_\delta^{\tau *}\text{+RN}) \models p = q * r.$

Proof. Let p, q, and r be closed $\mathrm{PA}_\delta^{\tau *}$+RN terms such that q is a guard; let $\mathcal{R}$ be a rooted branching bisimulation between p and $q \cdot p + r$. It can be shown that the following relation which uses the transitive closure of $\mathcal{R}$, denoted $\mathcal{R}^+$, is a rooted branching bisimulation between p and $q * r$:

$$\{(p, q * r)\} \cup \mathcal{R}^+ \cup \{(s, q * r) \mid s \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}\text{+RN}) \cup \{\surd\} \wedge s\mathcal{R}^+ p\}$$
$$\cup \{(s, t \cdot (q * r)) \mid s \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}\text{+RN}) \cup \{\surd\} \wedge t \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *}\text{+RN}) \wedge s\mathcal{R}^+ t \cdot p\}.$$

An explanation of this relation and the details of the proof can be found in (BASTEN AND VOORHOEVE, 1995a). □

Note that in the above proof a rooted branching bisimulation is constructed explicitly. In a process-algebraic theory with general recursion another proof strategy could be followed, namely the one used in (BAETEN AND WEIJLAND, 1990) and (BAETEN AND VERHOEF, 1995) to show that an operational semantics satisfies the general Recursive Specification Principle *RSP*. Any semantics satisfying *RSP* also satisfies *RSP**.

An example verification. To clarify the use of *FIR* and RSP^*, this paragraph contains an example verification using these two axioms. The calculations show how to verify an implementation of a system against its specification. The verification is an example of a typical application of process algebra.

Assume that the manager of a factory for the production of certain goods wants to have a production unit satisfying the following simple specification.

$$PU_{\text{spec}} = (imat \cdot omat) \,^*\, \delta. \tag{2.4.60}$$

Specification PU_{spec} states that the production unit performs two actions that are visible to the environment. First, it starts with getting input material (action *imat*). Second, after it has internally processed the input, it produces output material (*omat*). Of course, the input and output should satisfy certain requirements, but specification PU_{spec} abstracts away from such details. The iteration in the specification means that the unit should repeat the above two steps indefinitely (see Axioms *BKS*1 and *A*6).

A supplier has a production unit *PU* available of which the behavior can be described as follows.

$$\begin{aligned} PU = \; & (imat \cdot (ppmat^* fmat_1) \cdot (pmat_1 \parallel pmat_2) \cdot omat \cdot \\ & \quad imat \cdot (ppmat^* fmat_2) \cdot (pmat_1 \parallel pmat_2) \cdot omat \\ &) \,^*\, \delta. \end{aligned} \tag{2.4.61}$$

As required by the specification, unit *PU* performs a non-terminating iteration. Internally, the unit has two trays for unprocessed material. Each time the unit receives input material (*imat*), it performs a number of preprocessing steps (*ppmat*) and forwards the material to one of the two trays ($fmat_1, fmat_2$). Each processing step, the unit alternates between the two trays, starting with the first one. The number of preprocessing steps, if any at all, is always limited. The unit executes two processing actions on input material ($pmat_1$ and $pmat_2$). These processing actions can be done in parallel. If both processing actions are completed, the unit produces output material (*omat*). The supplier claims that the production unit satisfies the specification PU_{spec} of Property 2.4.60.

Of course, the factory manager wants to verify the claim. Since she is proficient in process algebra, she quickly performs a few calculations. Let I be the set of actions $\{ppmat, fmat_1, fmat_2, pmat_1, pmat_2\}$. Hiding these actions in the behavior *PU* should yield specification PU_{spec}. Formally, the proof requirement is specified as follows.

$$\tau_I(PU) = PU_{\text{spec}}. \tag{2.4.62}$$

The following derivation hides the actions in I in the behavior *PU* and eliminates the resulting silent steps. Note that it is allowed to use Axiom *FIR* because the number of preprocessing actions is limited.

$$\begin{aligned} & \tau_I(PU) \\ = \; & \quad \{ \text{Definition 2.4.61;} \\ & \quad\; TI1, 2, 4, TIM1, BKS4 \text{ (repeatedly)} \} \\ & (imat \cdot (\tau \,^*\, \tau) \cdot (\tau \parallel \tau) \cdot omat \cdot imat \cdot (\tau \,^*\, \tau) \cdot (\tau \parallel \tau) \cdot omat) \,^*\, \delta \\ = \; & \quad \{ FIR\ (2\times), M1, M2(2\times) \} \end{aligned}$$

$$
\begin{aligned}
& (imat \cdot (\tau + \tau \cdot \tau) \cdot (\tau \cdot \tau + \tau \cdot \tau) \cdot omat \cdot \\
& \quad imat \cdot (\tau + \tau \cdot \tau) \cdot (\tau \cdot \tau + \tau \cdot \tau) \cdot omat \\
&)^* \delta \\
= & \quad \{ B1, A3 \text{ (repeatedly)} \} \\
& (imat \cdot omat \cdot imat \cdot omat)^* \delta
\end{aligned}
$$

Summarizing,

$$\tau_I(PU) = (imat \cdot omat \cdot imat \cdot omat)^* \delta. \tag{2.4.63}$$

This is almost the desired result. Informally, the non-terminating iteration of some actions should be equal to the non-terminating iteration of twice these actions. Let $p \in \mathcal{C}(\mathrm{PA}_\delta^{\tau *} + \mathrm{RN})$ be a guard (see Definition 2.4.56 on Page 55).

$$
\begin{aligned}
& p^* \delta \\
= & \quad \{ BKS1(2\times) \} \\
& p \cdot (p \cdot (p^* \delta) + \delta) + \delta \\
= & \quad \{ A6, A5 \} \\
& (p \cdot p) \cdot (p^* \delta) + \delta
\end{aligned}
$$

Hence, it follows from RSP^* that

$$p^* \delta = (p \cdot p)^* \delta. \tag{2.4.64}$$

Property 2.4.64 proves the informal claim made above. Applying Property 2.4.64 to Result 2.4.63 easily yields the proof requirement of Property 2.4.62 on the preceding page.

$$\tau_I(PU) = PU_{\mathrm{spec}}. \tag{2.4.65}$$

Hence, the factory manager is satisfied and decides to take the production unit.

2.5 Conclusions

This chapter has given an introduction to basic concurrency theory. None of the material is new in itself. The main purpose of this chapter is to present it in a common framework. Thus, it provides a basis for the remaining chapters of this thesis.

Section 2.2 has introduced a very natural way to describe concurrent systems, namely transition systems. The notion of an operational semantics has been defined. This notion is elementary to the remainder of this thesis. Any formalism for describing concurrent systems introduced in this thesis is given a meaning in the framework of transition systems by providing it with an operational semantics.

In Section 2.3, the formalism of Place/Transition nets has been introduced. This formalism is a member of the large class of formalisms for modeling concurrent systems called Petri nets. Petri nets combine an intuitive graphical representation with strong analysis techniques. A class of Petri nets that is particularly suitable for the efficient analysis of properties of concurrent systems is the class of free-choice P/T nets.

Section 2.4 has given an introduction to ACP-style process algebras. Process algebras are symbolic formalisms for describing and analyzing concurrent-system behavior. They are particularly useful for the comparison of behaviors, for example, to verify an implementation against a specification.

The remainder of this thesis studies topics combining and integrating Petri nets and process algebra, using transition systems as a common framework.

3

Algebraic Verification of Modular Nets

3.1 Introduction

An essential property of any formalism for modeling and analyzing complex concurrent systems is *compositionality*. A formalism is compositional if it allows the system designer to structure a formal model of a system into component models of subsystems in such a way that properties of the system as a whole can be derived from the properties of the components. Unfortunately, labeled P/T nets as introduced in Chapter 2 are not suited for compositional design. They lack a structuring mechanism to divide models of concurrent systems into components. Process algebra, on the other hand, supports compositional design of concurrent systems. When the behavior of the subsystems of a concurrent system is described by means of algebraic terms, the behavior of the entire system can be described by composing these terms, using the suitable operators in the process algebra.

The goal of this chapter is to show how Petri nets and process algebra can be integrated into a formalism supporting compositional design of concurrent systems, combining the strengths of both formalisms. The resulting formalism is such that the combination of Petri nets and process algebra provides a better understanding of the system under development than any of the two formalisms in isolation.

The chapter is organized as follows. Section 3.2 briefly sketches a design method that provides the framework for the underlying integrated formalism. The design method emphasizes the strengths of both Petri nets and process algebra. Sections 3.3 through 3.6 present the formalism itself. The notion of modular P/T nets is introduced. Several algebraic semantics of modular P/T nets are given that allow a system designer to verify their behavior against an algebraic specification. The theory is a revised and improved version of the theory in (BASTEN AND VOORHOEVE, 1995b). To validate the design method and the underlying theory, in Sections 3.7 and 3.8, the results of two case studies are discussed. Finally, Section 3.9 presents some conclusions and open problems. It also contains a discussion of related work.

3.2 Design Method

First of all, it should be noted that it is not the intention of this section to present a complete design method that can be used for the design of real-world concurrent systems. The goal is merely to provide a framework for the formalism presented in the remaining sections.

In concurrent systems, the order in which communications between different parts of the system occur is often crucial. Petri nets and process algebra are both formalisms that focus on the dynamic behavior of systems. Therefore, the theory presented in this chapter is particularly useful for the design of concurrent systems where the communication protocol between the various parts of the system plays an important role. It is not very well suited for data-oriented applications or applications where most of the computations are performed by a single system component without much communication with other parts of the system. This does not mean that issues related to the data structures being used in the implementation of the system and the correctness of computations local to some specific part of the system are less important. However, other formalisms, such as assertional reasoning based on predicate logic and model checking techniques, are more useful for those purposes.

Petri nets and process algebra complement each other very well. Petri nets have an intuitive, graphical representation and are well suited to describe the dynamic behavior of a system including the states in which the system can be. Colored Petri nets, in particular, provide a strong formalism for modeling concurrent systems. Petri nets are very useful for purposes of system validation and simulation. In addition, the analysis techniques that are available for Petri nets can be used to show that a system satisfies some given property. Process algebra, on the other hand, is a compositional, purely symbolic formalism, designed to compare the dynamic behavior of different systems. By applying term rewriting techniques and equational reasoning, it can be verified whether an implementation satisfies a given specification, where both specification and implementation are algebraic terms.

Based on the above observations, a very simple design method is introduced that consists of four separate activities. Each activity is explained in more detail below.

i) Specify the behavior or behavioral properties of the concurrent system.

ii) Construct a (colored-)Petri-net model of the system.

iii) Use simulation to validate the specification and to test whether the Petri net conforms to the specification.

iv) Verify the correctness of the net with respect to its behavioral specification using process algebra.

Behavioral specification. In the early stages of system design, it is useful to specify in a concise way the order in which the actions in a concurrent system can occur. A process algebra or a simple specification language based on process-algebraic principles is well suited for this purpose. In this step, only *behavioral properties* are specified. Other details of the system are addressed when designing the Petri-net model of the system.

Petri-net model. Once a designer who is familiar with the method of this chapter has a clear understanding of the dynamic behavior of a system, he or she usually has no difficulty to construct a Petri net of which the behavior conforms to the specification of the system. For any non-trivial concurrent system, many more aspects than the order in which actions may occur are important. Examples of such aspects are the timing of actions, the data structures that are used in the implementation, and the algorithms needed for the local computations. These aspects are usually addressed in this stage of the design process. This means that, in general, a colored-Petri-net formalism is used to define the Petri-net model of the system.

Simulation. The Petri-net model of a concurrent system can be simulated using tools such as ExSpect (ASPT, 1994), Design/CPN (JENSEN ET AL., 1991), or PEP (BEST AND GRAHLMANN, 1995). The specification and the net model can be validated and corrected, if necessary. Simulation is an excellent means to discover the more obvious mistakes in both the behavioral specification and the Petri-net model. However, by means of simulation only, it is usually impossible to guarantee that all errors are discovered.

Note that simulation can also be used for other purposes, such as performance analysis. Although performance analysis may be very useful for many applications, it is not an important activity in the design method proposed here, which focuses on behavioral correctness.

Verification. In the verification phase, the Petri-net model and, if necessary, also the behavioral specification are formally translated into a process-algebra expression. By means of term rewriting techniques and equational reasoning, it is verified whether the behavior of the net conforms to its specification. This step is useful to discover the more subtle mistakes in a specification and the corresponding Petri net.

The presentation of the four activities suggests an ordering. In an ideal situation, they are indeed applied in the order presented. However, the experience of the case studies presented in Sections 3.7 and 3.8 shows that, in practice, they are seldom clearly separated. Systems design is a complex process where one often works on several of the four activities simultaneously. If one discovers an error during one step, it may be necessary to redo some of the work in other steps. Even more important, if one follows a top-down design strategy, it is possible to first design and verify a system on a high level of abstraction before adding more detail. When adding a detailed level of abstraction, the design cycle is repeated.

The next four sections present basic theoretical results, integrating Petri nets and process algebra, that are needed in the modeling and verification phases of the above design method. Although in practice, a colored-Petri-net formalism is needed, the results in this chapter are restricted to an uncolored formalism, called modular P/T nets. Modular P/T nets extend P/T nets as defined in Chapter 2 with a hierarchy construct. As mentioned, such a construct is essential in compositional design. The extension of the theory to colored Petri nets remains for future investigation. In Sections 3.7 and 3.8, where two case studies are discussed, a very simple language for behavioral specifications is introduced. In these sections, the role of simulation in the method is also explained in more detail.

3.3 P/T-Net Components

The basis of a modular P/T net is the so-called P/T-net *component*. P/T-net components are the building blocks used to construct a modular P/T net. Figure 3.3.1 shows an example of a P/T-net component. P/T-net components strongly resemble standard P/T nets as introduced in Chapter 2. The only difference is that, for P/T-net components, the set of places is divided into *internal places* and *pins*. The graphical representation of a P/T-net component uses a dashed box to divide the set of places into internal places and pins. Pins are depicted as double circles. The idea is that pins are connectors to the environment. The environment can interact with the P/T-net component by removing tokens from or putting tokens into its pins. Since transition labels do not play an important role in the theory of this chapter, *unlabeled* P/T nets, as defined in Definition 2.3.39 on Page 26, are taken as the starting point for this chapter. The extension with transition labels is straightforward.

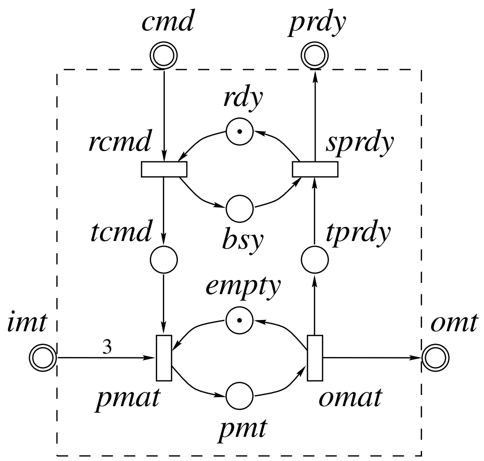


Figure 3.3.1: The graphical representation of a P/T-net component.

Definition 3.3.2. (P/T-net component) A P/T-net component is a pair $C = (N, I)$, where $N = (P, T, F, W)$ is a P/T net as defined in Definition 2.3.39 on Page 26 and where $I \subseteq P$ is the set of internal places.

For any P/T-net component C, the set $P \backslash I$ is the set of pins of the component, also called the *interface* of the component, denoted $\mathit{ifc}(C)$. For any transition $t \in T$, its bag of *internal input places* $\boldsymbol{iit}$ is defined as $\boldsymbol{it} \lceil I$; its bag of *observable input places* $\boldsymbol{oit}$ is defined as $\boldsymbol{it} \lceil \mathit{ifc}(C)$. The bags of internal and observable output places, denoted $\boldsymbol{iot}$ and $\boldsymbol{oot}$, respectively, are defined likewise.

Example 3.3.3. Figure 3.3.1 shows an example of a P/T-net component modeling a production unit. Its interface is the set of pins $\{\mathit{cmd}, \mathit{prdy}, \mathit{imt}, \mathit{omt}\}$. The bag of input places of transition *rcmd*, ***i**rcmd*, is equal to [*cmd*, *rdy*]; its bag of output places ***o**rcmd* is equal to [*tcmd*, *bsy*]. Consequently, its internal input places ***ii**rcmd* and its internal output places ***io**rcmd* are the bags [*rdy*] and [*tcmd*, *bsy*], respectively; its bag of observable input places ***oi**rcmd* is the singleton bag [*cmd*] and the bag of its observable output places ***oo**rcmd* is the empty bag **0**.

Definition 3.3.4. (Marked P/T-net component) A *marked* P/T-net component is a triple (N, I, s), where (N, I) is a P/T-net component as defined above and where s is a bag over I denoting the *internal marking* of the net component. The set of all marked P/T-net components is denoted $\mathcal{C}$.

Pins are not considered in the marking of a P/T-net component, because the goal is to determine the behavior of such a component under the assumption that the environment is responsible for consuming tokens from and producing tokens on pins. It is assumed that the environment has always tokens available when they are needed by a P/T-net component. It is also assumed that the environment is always willing to receive any tokens that the component might produce. Therefore, transition enabling is defined in such a way that it does not depend on pins.

Definition 3.3.5. (Transition enabling) Let (N, I, s) be a marked P/T-net component, where $N = (P, T, F, W)$. A transition $t \in T$ is *enabled*, denoted $(N, I, s)[t\rangle$, if and only if each *internal* input place p of t contains at least as many tokens as its cardinality in $\boldsymbol{ii}t$. That is, $(N, I, s)[t\rangle \Leftrightarrow \boldsymbol{ii}t \leq s$.

Example 3.3.6. Consider again the marked P/T-net component in Figure 3.3.1 on the facing page. Its pins *cmd*, *prdy*, *imt*, and *omt* do not contain any tokens, since only internal places can be marked in a P/T-net component. In the initial marking depicted in the figure, only transition *rcmd* is enabled.

The firing rule for P/T-net components is very similar to the firing rule for labeled P/T nets (see Definition 2.3.7 on Page 18). There are two differences. The first one is that the firing rule for P/T-net components only affects tokens in internal places. The second difference is the action function. As already mentioned, the environment and a P/T-net component interact via the pins of the component. From the point of view of the environment, it is therefore most appropriate to define the behavior of a P/T-net component in terms of the production and consumption of tokens. In a single action, any number of tokens can be consumed or produced. This means that an action is uniquely determined by two bags of tokens, one denoting the consumed tokens and one denoting the produced tokens. As a result, actions are no longer atomic, as in the previous chapter, but they are *structured*. For this reason, actions of P/T-net components are also called *multi-actions*. Recall that transition names, place names, and hence also token identifiers, are taken from the universe of identifiers U introduced in Section 2.3.2.

Definition 3.3.7. (Multi-actions) The set of multi-actions $\boldsymbol{A}$ is defined as $\mathcal{B}(U) \times \mathcal{B}(U)$, the set of all pairs of bags over U.

Similar to Section 2.3.2, for any P/T-net component $C = (N, I)$ with $N = (P, T, F, W)$, an action function $\alpha_C : T \to \boldsymbol{A}$ is a function assigning to each transition of C a corresponding multi-action. Since it is always clear from the context to which P/T-net component an action function refers, in the remainder, the subscript of action functions is omitted.

Definition 3.3.8. (Firing rule) The firing rule $\_\,[\_\rangle\,\_ \subseteq \mathcal{C} \times \boldsymbol{A} \times \mathcal{C}$ is the smallest relation satisfying for any (N, I, s) in $\mathcal{C}$, with $N = (P, T, F, W)$ and action function $\alpha : T \to \boldsymbol{A}$, and any $t \in T$,

$(N, I, s)[t\rangle \Rightarrow (N, I, s)\ [\alpha(t)\rangle\ (N, I, s - \boldsymbol{iit} \uplus \boldsymbol{iot})$.

When analyzing the behavior of a P/T-net component in isolation, it is useful to be able to look inside the component as though it were a *glass box*. This means that all consumptions and productions of tokens that result from firing a transition are visible. When using a component in a compositional design, on the other hand, it is often useful to assume that its internal structure is hidden as in a *black box*. The environment observes the actions of the P/T-net component projected onto its pins. When formalizing the semantics for P/T-net components, the action function in the definition of the firing rule determines which point of view is taken.

Definition 3.3.9. (Action functions) Let (N, I) with $N = (P, T, F, W)$ be a P/T-net component.

Glass-box action function: For any transition $t \in T$, $\alpha^g(t)$ is equal to $(\boldsymbol{it}, \boldsymbol{ot})$.

Black-box action function: For any transition $t \in T$, $\alpha^b(t)$ is defined as $(\boldsymbol{oit}, \boldsymbol{oot})$.

As for labeled P/T nets, the firing rule is the basis for the operational semantics for P/T-net components. The semantics defined in this chapter do not distinguish between successful termination and deadlock. In this respect, they adhere to the usual semantics given for Petri nets (see also Section 2.3.2). The following definition gives the glass-box semantics for P/T-net components.

Definition 3.3.10. (Operational glass-box semantics) The glass-box behavior of marked P/T-net components is formalized by the operational semantics $\mathcal{S}^g(\mathcal{C})$. It consists of the process space $(\mathcal{C}, \boldsymbol{A},\ [\ \rangle\ , \emptyset)$, where the firing rule is instantiated with the glass-box action function α^g. The equivalence on processes is strong bisimilarity.

$$\begin{tikzcd} & s_0 \arrow[dl, "rcmd"'] & \\ s_1 \arrow[dr, "pmat"'] & & s_3 \arrow[ul, "sprdy"'] \\ & s_2 \arrow[ur, "omat"'] & \end{tikzcd}$$

$s_0 = [rdy, empty]$
$s_1 = [bsy, tcmd, empty]$
$s_2 = [bsy, pmt]$
$s_3 = [bsy, tprdy, empty]$

Figure 3.3.11: The operational semantics of the production-unit component.

Example 3.3.12. Figure 3.3.11 on the facing page shows the operational glass-box semantics for the P/T-net component of Figure 3.3.1 on Page 62. The firing rule is depicted by arrows. Processes are identified with their markings. The transition identifier t is used as an abbreviation of the corresponding multi-action $\alpha^g(t)$. For example, *rcmd* is an abbreviation of ([*cmd*, *rdy*], [*tcmd*, *bsy*]).

The next definition gives the black-box semantics for P/T-net components. As may be expected, it uses rooted branching bisimilarity as the equivalence relation. It is important to note that the silent multi-action in $\boldsymbol{A}$ is the pair $(\mathbf{0}, \mathbf{0})$.

Definition 3.3.13. (Operational black-box semantics) The black-box behavior of P/T-net components is formalized by the operational semantics $\mathcal{S}^b(\mathcal{C})$. It consists of the process space $(\mathcal{C}, \boldsymbol{A}, [\,\rangle, \emptyset)$, where the firing rule is instantiated with the black-box action function α^b. The equivalence on processes is rooted branching bisimilarity.

Example 3.3.14. Consider again Figure 3.3.11 on the facing page. Let the identifier t of a transition be an abbreviation of $\alpha^b(t)$, which is the pair $(\boldsymbol{oit}, \boldsymbol{oot})$. For example, *rcmd* is an abbreviation of ([*cmd*], $\mathbf{0}$). Under this assumption, Figure 3.3.11 shows the semantics for the P/T-net component of Figure 3.3.1 on Page 62 in the black-box semantics $\mathcal{S}^b(\mathcal{C})$.

3.4 Algebraic Semantics for P/T-Net Components

In the previous section, two different operational semantics for P/T-net components have been defined, namely the operational glass-box semantics and the operational black-box semantics. This section presents two algebraic semantics for P/T-net components. It is shown that the algebraic semantics are consistent with the two operational semantics of the previous section.

3.4.1 Glass-box semantics

The equational theory $\mathrm{PA}^*_{\delta\lambda}$. The algebraic glass-box semantics for P/T-net components is defined in an extension of the process algebra PA^*_{δ}, Process Algebra with inaction and iteration. The theory PA^*_{δ} is defined by combining the appropriate ingredients introduced in Chapter 2. It is extended with the so-called state operator λ yielding the theory $\mathrm{PA}^*_{\delta\lambda}$, for Process Algebra with inaction, state operator, and iteration. The state operator is an operator which has a memory to explicitly describe the state of a process. For a detailed treatment of the state operator, see (BAETEN AND VERHOEF, 1995; BAETEN AND WEIJLAND, 1990). The state operator used to define an algebraic semantics for P/T-net components is called the *causal state operator*. A variant of the causal state operator first appeared in (BAETEN AND BERGSTRA, 1993). Table 3.4.1 on the following page presents the theory $\mathrm{PA}^*_{\delta\lambda}$ which is as always parameterized by a set of actions. It is instantiated with the set of multi-actions $\boldsymbol{A}$ defined in the previous section (Definition 3.3.7 on Page 63). The following auxiliary functions on multi-actions in $\boldsymbol{A}$ are defined. For any action $a = (c, p)$ in $\boldsymbol{A}$, $\boldsymbol{ca}$ is equal to c and defines the *consumption* of a; $\boldsymbol{pa}$ is p and defines the *production* of a.

$\mathrm{PA}^*_{\delta\lambda}(\boldsymbol{A})$	
$\mathrm{PA}^*_{\delta}(\boldsymbol{A})$	
$\lambda^{\_}_{\_} : (\mathcal{P}(U) \times \mathcal{B}(U)) \to (\boldsymbol{P} \to \boldsymbol{P})$	
$a : \boldsymbol{A};\ I : \mathcal{P}(U);\ s : \mathcal{B}(U);\ x, y : \boldsymbol{P};$	
$\lambda^I_s(\delta) = \delta$	$CSO1$
$\boldsymbol{ca} \lceil I \leq s \Rightarrow \lambda^I_s(a) = a$	$CSO2$
$\boldsymbol{ca} \lceil I \not\leq s \Rightarrow \lambda^I_s(a) = \delta$	$CSO3$
$\lambda^I_s(a \cdot x) = \lambda^I_s(a) \cdot \lambda^I_{s-\boldsymbol{ca}\lceil I \uplus \boldsymbol{pa}\lceil I}(x)$	$CSO4$
$\lambda^I_s(x + y) = \lambda^I_s(x) + \lambda^I_s(y)$	$CSO5$

Table 3.4.1: The process algebra $\mathrm{PA}^*_{\delta\lambda}$.

For I, s, and x as in Table 3.4.1, the term $\lambda^I_s(x)$ can be thought of as the P/T-net component x with internal places I and marking s. The operator is axiomatized by means of five axioms. Axioms $CSO2$, $CSO3$, and $CSO4$ are the most interesting ones. The other two should be clear without further explanation. Axiom $CSO2$ can be read as follows. It states that an action may occur provided that its consumption from *internal* places is available in the marking. Axiom $CSO3$ says that if not enough tokens are available, an action cannot be executed and results in a deadlock. Axiom $CSO4$ states that the result of executing an action is that the consumption is removed from the marking, whereas the production in internal places is added to the marking. If the marking does not contain enough tokens to execute the action, then the combination of Axioms $CSO3$ and $A7$ guarantees that the result is a deadlock. Note the correspondence between the causal state operator and the firing rule for P/T-net components.

An operational semantics for $\mathrm{PA}^*_{\delta\lambda}$. The theory $\mathrm{PA}^*_{\delta\lambda}$ is given an operational semantics in the standard way (see Section 2.4). The processes in the operational semantics for the theory $\mathrm{PA}^*_{\delta\lambda}$ are the closed $\mathrm{PA}^*_{\delta\lambda}$ terms extended with the special process $\surd$. The transition relation $\_ \xrightarrow{\_} \_ \subseteq (\mathcal{C}(\mathrm{PA}^*_{\delta\lambda}) \cup \{\surd\}) \times \boldsymbol{A} \times (\mathcal{C}(\mathrm{PA}^*_{\delta\lambda}) \cup \{\surd\})$ is the smallest relation satisfying the appropriate derivation rules given in Tables 2.4.5 on Page 36, 2.4.24 on Page 43, 2.4.48 on Page 53, and 3.4.2 on the next page, where in all tables p, p', q, and q' range over $\mathcal{C}(\mathrm{PA}^*_{\delta\lambda})$.

Definition 3.4.3. (Operational semantics) The operational semantics $\mathcal{S}(\mathrm{PA}^*_{\delta\lambda})$ is the process space $(\mathcal{C}(\mathrm{PA}^*_{\delta\lambda}) \cup \{\surd\}, \boldsymbol{A}, \longrightarrow, \{\surd\})$ with strong bisimilarity as the equivalence relation.

Property 3.4.4. (Congruence) *Strong bisimilarity, $\sim$, is a congruence for the operators of* $\mathrm{PA}^*_{\delta\lambda}$.

$a : \boldsymbol{A};\ I : \mathcal{P}(U);\ s : \mathcal{B}(U);\ p, p' : \mathcal{C}(\mathrm{PA}^*_{\delta\lambda});$

$$\frac{p \xrightarrow{a} p'}{\lambda^I_{s \uplus \boldsymbol{ca}\restriction I}(p) \xrightarrow{a} \lambda^I_{s \uplus \boldsymbol{pa}\restriction I}(p')} \qquad \frac{p \xrightarrow{a} \surd}{\lambda^I_{s \uplus \boldsymbol{ca}\restriction I}(p) \xrightarrow{a} \surd}$$

Table 3.4.2: The transition relation for the causal state operator.

Proof. It follows from the format of the derivation rules in Tables 2.4.5, 2.4.24, 2.4.48, and 3.4.2 (see BAETEN AND VERHOEF, 1995). □

Theorem 3.4.5. (Soundness) *For any closed terms* $p, q \in \mathcal{C}(\mathrm{PA}^*_{\delta\lambda})$,
$\mathrm{PA}^*_{\delta\lambda} \vdash p = q \Rightarrow \boldsymbol{\mathcal{S}}(\mathrm{PA}^*_{\delta\lambda}) \models p = q.$

Proof. Since bisimilarity is a congruence for the operators of $\mathrm{PA}^*_{\delta\lambda}$, it is sufficient to show the validity of each of the axioms of $\mathrm{PA}^*_{\delta\lambda}$. It is not difficult to construct a bisimulation for each case. □

As explained in Section 2.4.5, there is no completeness result for $\mathrm{PA}^*_{\delta\lambda}$.

Standard concurrency and expansion. The theory $\mathrm{PA}^*_{\delta\lambda}$ can be extended with the axioms of standard concurrency in exactly the same way as in Section 2.4.5. It is not difficult to see that the rooted branching bisimulations given in the proof of Property 2.4.53 on Page 53 are also strong bisimulations.

Recursive equations. The theory $\mathrm{PA}^*_{\delta\lambda}$ is extended with the principle *RSP** in the same way as in Section 2.4.5.

An algebraic semantics for P/T-net components. At this point, all ingredients needed to define an algebraic semantics for P/T-net components are present. Each P/T-net component is translated into a closed $\mathrm{PA}^*_{\delta\lambda}$ term with the same operational behavior as the component, assuming that the glass-box semantics for P/T-net components is chosen. The idea is to start by defining the unrestricted behavior of a component. That is, its behavior when every transition is always enabled. Then, the causal state operator instantiated with the marking of the component is used to restrict the behavior to all possible firing sequences. The unrestricted behavior of a single transition is the non-terminating iteration of its consumption and production of tokens. The unrestricted behavior of a net component is the parallel execution of all its transitions.

Definition 3.4.6. (Algebraic glass-box semantics) Let $C = (N, I, s)$ be a P/T-net component in $\boldsymbol{\mathcal{C}}$, where $N = (P, T, F, W)$. The algebraic glass-box semantics for C, denoted $\mathcal{C}^g[\![C]\!]$, is defined as follows:
$\mathcal{C}^g[\![C]\!] = \lambda^I_s(\| \, t : t \in T : \alpha^g(t) * \delta),$
where α^g is the glass-box action function defined in Definition 3.3.9 on Page 64. Note that

α^g is not an operator of the theory $\mathrm{PA}^*_{\delta\lambda}$. Hence, $\alpha^g(t)$, where t is a transition in T, is not a term in the algebra. However, the value of $\alpha^g(t)$ is a multi-action in $\boldsymbol{A}$ and, hence, a term in $\mathrm{PA}^*_{\delta\lambda}$. Therefore, in the above definition, $\alpha^g(t)$ should be interpreted as the function value and not as a syntactical term.

Example 3.4.7. Let C be the P/T-net component of Figure 3.3.1 on Page 62. As before, the identifier of a transition t of C is used as an abbreviation of $\alpha^g(t)$. Let I be the set of internal places of C, $\{rdy, bsy, tcmd, tprdy, empty, pmt\}$. The algebraic glass-box semantics of C is now as follows:

$$\lambda^I_{[rdy,empty]}(rcmd^*\delta \parallel pmat^*\delta \parallel sprdy^*\delta \parallel omat^*\delta).$$

Assuming the glass-box semantics for P/T-net components, a component C and its algebraic representation $\mathcal{C}^g[\![C]\!]$ have the same operational semantics. That is, any action that a component can perform can be simulated by its algebraic semantics and vice versa. Furthermore, since a component cannot terminate successfully, its algebraic semantics cannot terminate successfully either. This means that the algebraic semantics of a net component cannot evolve into the special process $\surd$. In other words, in a process space combining P/T-net components and closed $\mathrm{PA}^*_{\delta\lambda}$ terms, a P/T-net component and its algebraic semantics are strongly bisimilar. Recall that $\equiv$ denotes syntactical equality of closed terms.

Theorem 3.4.8. *Assume the semantics for P/T-net components is the operational glass-box semantics $\mathcal{S}^g(\mathcal{C})$ of Definition 3.3.10 on Page 64. For any P/T-net components (N, I, s) and (N, I, s') in $\mathcal{C}$, closed term p in $\mathcal{C}(\mathrm{PA}^*_{\delta\lambda})$, and action $a \in \boldsymbol{A}$,*

i) $(N, I, s)\ [a\rangle\ (N, I, s') \Rightarrow \mathcal{C}^g[\![(N, I, s)]\!] \xrightarrow{a} \mathcal{C}^g[\![(N, I, s')]\!]$,

ii) $\mathcal{C}^g[\![(N, I, s)]\!] \xrightarrow{a} p \Rightarrow$
$(\exists s' : s' \in \mathcal{B}(I) : p \equiv \mathcal{C}^g[\![(N, I, s')]\!] \wedge (N, I, s)\ [a\rangle\ (N, I, s'))$,

iii) $\mathcal{C}^g[\![(N, I, s)]\!] \not\xrightarrow{a} \surd$.

Proof. Let N be equal to (P, T, F, W).

i) Assume $(N, I, s)\ [a\rangle\ (N, I, s')$. It follows from the definitions of the semantics $\mathcal{S}^g(\mathcal{C})$ for P/T-net components (Definition 3.3.10 on Page 64) and the firing rule (Definition 3.3.8 also on Page 64) that there must be a transition u in T, enabled in marking s, such that $a = \alpha^g(u)$ and $s' = s - \boldsymbol{iiu} \uplus \boldsymbol{iou}$. Definition 3.3.5 (Transition enabling) on Page 63 yields that $\boldsymbol{iiu} \le s$. It follows from Definition 3.3.9 (Action function α^g) on Page 64 that $\boldsymbol{ca} = \boldsymbol{iu}$ and $\boldsymbol{pa} = \boldsymbol{ou}$. Hence, $\boldsymbol{ca} \upharpoonright I = \boldsymbol{iiu}$ and $\boldsymbol{pa} \upharpoonright I = \boldsymbol{iou}$. Consequently, $s' = s - \boldsymbol{iiu} \uplus \boldsymbol{iou} = s - \boldsymbol{ca} \upharpoonright I \uplus \boldsymbol{pa} \upharpoonright I$. The following steps follow from the derivation rules in Tables 2.4.5 on Page 36, 2.4.24 on Page 43, 2.4.48 on Page 53, and 3.4.2 on the previous page. The fact that

$$a \xrightarrow{a} \surd$$

implies that

$$a^*\delta \xrightarrow{a} a^*\delta.$$

Since $u \in T$ and $a = \alpha^g(u)$,

$$(\parallel t : t \in T : \alpha^g(t)^*\delta) \xrightarrow{a} (\parallel t : t \in T : \alpha^g(t)^*\delta).$$

It follows from the expression derived above for s' that
$\lambda^I_s(\| t : t \in T : \alpha^g(t)^*\delta) \xrightarrow{a} \lambda^I_{s'}(\| t : t \in T : \alpha^g(t)^*\delta)$.
Hence, Definition 3.4.6 (Algebraic semantics) on Page 67 yields
$\mathcal{C}^g[\![(N, I, s)]\!] \xrightarrow{a} \mathcal{C}^g[\![(N, I, s')]\!]$,
which concludes the first part of the proof.

ii) Assume $\mathcal{C}^g[\![(N, I, s)]\!] \xrightarrow{a} p$. Definition 3.4.6 (Algebraic semantics) on Page 67 yields
$\lambda^I_s(\| t : t \in T : \alpha^g(t)^*\delta) \xrightarrow{a} p$.
Recall that the transition relation $\longrightarrow$ in the operational semantics $\boldsymbol{\mathcal{S}}(\mathrm{PA}^*_{\delta\lambda})$ (Definition 3.4.3 on Page 66) is defined as the *smallest* relation satisfying a set of derivation rules. Therefore, it follows from the derivation rules for the causal state operator in Table 3.4.2 on Page 67 that process $\lambda^I_s(\| t : t \in T : \alpha^g(t)^*\delta)$ can only perform an a action if there is a closed term $q \in \mathcal{C}(\mathrm{PA}^*_{\delta\lambda})$ such that $\boldsymbol{ca} \restriction I \leq s$,
$p \equiv \lambda^I_{s-\boldsymbol{ca}\restriction I \uplus \boldsymbol{pa}\restriction I}(q)$,
and
$(\| t : t \in T : \alpha^g(t)^*\delta) \xrightarrow{a} q$.
In the following, empty merge quantifications must be omitted. In addition, for the sake of simplicity, closed terms that are equal up to symmetry and associativity of the merge are considered syntactically equal. The derivation rules for the merge operator in Table 2.4.24 on Page 43 yield that there must be a transition $u \in T$ and a closed term $r \in \mathcal{C}(\mathrm{PA}^*_{\delta\lambda})$ such that
$q \equiv r \,\|\, (\| t : t \in T \wedge t \neq u : \alpha^g(t)^*\delta)$
and
$\alpha^g(u)^*\delta \xrightarrow{a} r$.
It follows from the derivation rules for the Kleene star in Table 2.4.48 on Page 53 that $a = \alpha^g(u)$ and
$r \equiv \alpha^g(u)^*\delta$.
Hence, it follows from the definition of α^g (Definition 3.3.9 on Page 64) that $\boldsymbol{ca} = \boldsymbol{iu}$ and $\boldsymbol{pa} = \boldsymbol{ou}$. Therefore, $s - \boldsymbol{ca} \restriction I \uplus \boldsymbol{pa} \restriction I = s - \boldsymbol{iiu} \uplus \boldsymbol{iou}$. Summarizing the results so far yields
$p \equiv \lambda^I_{s-\boldsymbol{iiu}\uplus\boldsymbol{iou}}(\| t : t \in T : \alpha^g(t)^*\delta)$,
which by Definition 3.4.6 (Algebraic semantics) on Page 67 means that
$p \equiv \mathcal{C}^g[\![(N, I, s - \boldsymbol{iiu} \uplus \boldsymbol{iou})]\!]$.
Since $\boldsymbol{ca} \restriction I \leq s$, it follows that $\boldsymbol{iiu} \leq s$. It follows from the definition of transition enabling for P/T-net components (Definition 3.3.5 on Page 63), the definition of the firing rule (Definition 3.3.8 on Page 64), and the fact that $\alpha^g(u) = a$ that
$(N, I, s)\ [a\rangle\ (N, I, s - \boldsymbol{iiu} \uplus \boldsymbol{iou})$,
which completes the second part of the proof.

iii) Assume $\mathcal{C}^g[\![(N, I, s)]\!] \xrightarrow{a} \surd$. It is straightforward to derive from Definition 3.4.6 on Page 67 (Algebraic semantics) and the derivation rules in Table 3.4.2 also on Page 67 that
$(\| t : t \in T : \alpha^g(t)^*\delta) \xrightarrow{a} \surd$.

It follows from the derivation rules in Table 2.4.24 on Page 43 that this is only possible if T is a singleton set $\{u\}$, for some transition identifier $u \in U$, such that
$\alpha^g(u) * \delta \xrightarrow{a} \surd$.
The derivation rules for the binary Kleene star in Table 2.4.48 on Page 53 yield that
$\delta \xrightarrow{a} \surd$.
Since this clearly is a contradiction,
$\mathcal{C}^g[\![(N, I, s)]\!] \not\xrightarrow{a} \surd$,
which concludes the proof. □

The following property is a simple variant of the expansion theorem, presented in Section 2.4.3, which is often useful in verifications concerning the algebraic semantics of P/T-net components.

Property 3.4.9. *For any non-empty bag of actions $B \in \mathcal{B}(A)$,*
$(\| a : a \in B : a * \delta) = (+ b : b \in B : b \cdot (\| a : a \in B : a * \delta))$.

Proof. If B is a singleton bag containing only one action a, then the above equation reduces to $a * \delta = a \cdot (a * \delta)$, which follows immediately from Axioms $BKS1$ and $A6$. So, assume that B contains at least two different actions.

$$
\begin{array}{ll}
 & (\| a : a \in B : a * \delta) \\
= & \quad \{ \text{Expansion} \} \\
 & (+ b : b \in B : b * \delta \mathbin{\|\!\lfloor} (\| a : a \in B \wedge a \neq b : a * \delta)) \\
= & \quad \{ BKS1, A6 \} \\
 & (+ b : b \in B : b \cdot (b * \delta) \mathbin{\|\!\lfloor} (\| a : a \in B \wedge a \neq b : a * \delta)) \\
= & \quad \{ M3 \} \\
 & (+ b : b \in B : b \cdot ((b * \delta) \| (\| a : a \in B \wedge a \neq b : a * \delta))) \\
= & \quad \{ ASC6 \text{ (associativity of merge)} \} \\
 & (+ b : b \in B : b \cdot (\| a : a \in B : a * \delta))
\end{array}
$$
□

Example 3.4.10. Consider again Example 3.4.7 on Page 68. The following derivations yield a simple algebraic expression for the behavior of the P/T-net component C of Figure 3.3.1 on Page 62. The same abbreviations are used as in Example 3.4.7. In addition, let X denote the term $rcmd * \delta \| pmat * \delta \| sprdy * \delta \| omat * \delta$. The superscript I of the causal state operator, denoting the internal places of component C, is omitted.

$$
\begin{array}{ll}
 & \mathcal{C}^g[\![C]\!] \\
= & \quad \{ \text{Definition 3.4.6 (Algebraic semantics) on Page 67} \} \\
 & \lambda_{[rdy,empty]}(X) \\
= & \quad \{ \text{Property 3.4.9} \} \\
 & \lambda_{[rdy,empty]}(rcmd \cdot X + pmat \cdot X + sprdy \cdot X + omat \cdot X) \\
= & \quad \{ CSO5 \text{ (repeatedly)} \} \\
 & \lambda_{[rdy,empty]}(rcmd \cdot X) + \lambda_{[rdy,empty]}(pmat \cdot X) + \lambda_{[rdy,empty]}(sprdy \cdot X) + \lambda_{[rdy,empty]}(omat \cdot X) \\
= & \quad \{ CSO4 \text{ (repeatedly)} \} \\
 & \lambda_{[rdy,empty]}(rcmd) \cdot \lambda_{[bsy,tcmd,empty]}(X) + \lambda_{[rdy,empty]}(pmat) \cdot \lambda_{[rdy,pmt]}(X) + \\
 & \quad \lambda_{[rdy,empty]}(sprdy) \cdot \lambda_{[rdy^2,empty]}(X) + \lambda_{[rdy,empty]}(omat) \cdot \lambda_{[rdy,tprdy,empty^2]}(X)
\end{array}
$$

$=$ { $CSO2$, $CSO3$ (repeatedly), A7 (repeatedly) }
$rcmd \cdot \lambda_{[bsy,tcmd,empty]}(X) + \delta + \delta + \delta$
$=$ { $A6$ (repeatedly) }
$rcmd \cdot \lambda_{[bsy,tcmd,empty]}(X)$

Derivations similar to this one yield the following results.

$\lambda_{[bsy,tcmd,empty]}(X) = pmat \cdot \lambda_{[bsy,pmt]}(X)$,
$\lambda_{[bsy,pmt]}(X) = omat \cdot \lambda_{[bsy,tprdy,empty]}(X)$, and
$\lambda_{[bsy,tprdy,empty]}(X) = sprdy \cdot \lambda_{[rdy,empty]}(X)$.

Summarizing,

$$\lambda_{[rdy,empty]}(X) = rcmd \cdot pmat \cdot omat \cdot sprdy \cdot \lambda_{[rdy,empty]}(X).$$

Since clearly $rcmd \cdot pmat \cdot omat \cdot sprdy$ is a guard (see Definition 2.4.56 on Page 55), RSP^* and $A6$ yield

$$\lambda_{[rdy,empty]}(X) = (rcmd \cdot pmat \cdot omat \cdot sprdy) * \delta.$$

Concluding,

$$\mathcal{C}^g[\![C]\!] = (rcmd \cdot pmat \cdot omat \cdot sprdy) * \delta.$$

It is not difficult to see that the operational semantics for $\mathcal{C}^g[\![C]\!]$ is identical to the process depicted in Figure 3.3.11 on Page 64. Hence, Theorem 3.4.8 on Page 68 is clearly satisfied.

3.4.2 Black-box semantics

The equational theory $\mathrm{PA}^{\tau *}_{\delta\lambda}$+RN. In order to define the algebraic black-box semantics for P/T-net components, the process algebra $\mathrm{PA}^{*}_{\delta\lambda}$, defined in the previous subsection, is extended with silent actions and renaming. The result is the process algebra $\mathrm{PA}^{\tau *}_{\delta\lambda}$+RN, Process Algebra with inaction, state operator, silent actions, iteration, and renaming. It is parameterized with a set of actions, which is instantiated with the set of multi-actions $\boldsymbol{A}$ (see Definition 3.3.7 on Page 63).

In $\mathrm{PA}^{\tau *}_{\delta\lambda}$+RN, it is possible to define an abstraction operator similar to the operator defined in Section 2.4.4. It is slightly different, because the multi-actions in the current setting are structured. Nevertheless, the standard notation τ_I is used, where $I \subseteq U$ is a set of identifiers. The renaming function $\boldsymbol{f}_I : \boldsymbol{A} \cup \{\delta\} \rightarrow \boldsymbol{A} \cup \{\delta\}$ that is the basis for the abstraction operator is defined as follows: $\boldsymbol{f}_I(\delta) = \delta$; for any $a \in \boldsymbol{A}, \boldsymbol{f}_I(a) = (\boldsymbol{ca} \upharpoonright (U \backslash I), \boldsymbol{pa} \upharpoonright (U \backslash I))$. Note that this definition satisfies the requirement for the silent action $(\mathbf{0}, \mathbf{0})$ that $\boldsymbol{f}_I(\mathbf{0}, \mathbf{0}) = (\mathbf{0}, \mathbf{0})$. Table 3.4.11 on the next page gives the axioms for the abstraction operator corresponding to Axioms $RN1$ through $RN3$, $BKS4$, $RNM1$, and $RNM2$ for the general renaming operator.

An operational semantics for $\mathrm{PA}^{\tau *}_{\delta\lambda}$+RN. The processes in the operational semantics for $\mathrm{PA}^{\tau *}_{\delta\lambda}$+RN are the closed $\mathrm{PA}^{\tau *}_{\delta\lambda}$+RN terms extended with the special process $\surd$. The transition relation $\_ \xrightarrow{\_} \_ \subseteq (\mathcal{C}(\mathrm{PA}^{\tau *}_{\delta\lambda}\text{+RN}) \cup \{\surd\}) \times \boldsymbol{A} \times (\mathcal{C}(\mathrm{PA}^{\tau *}_{\delta\lambda}\text{+RN}) \cup \{\surd\})$ is the smallest relation satisfying the derivation rules given in Tables 2.4.5 on Page 36, 2.4.24

$a : \boldsymbol{A};\ I : \mathcal{P}(U);\ x, y : \boldsymbol{P};$	
$\tau_I(\delta) = \delta$	$TA1$
$\tau_I(a) = (\boldsymbol{ca} \upharpoonright (U \backslash I), \boldsymbol{pa} \upharpoonright (U \backslash I))$	$TA2$
$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	$TA3$
$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$	$TA4$
$\tau_I(x \,{}^*\, y) = \tau_I(x) \,{}^*\, \tau_I(y)$	$TA5$
$\tau_I(x \parallel y) = \tau_I(x) \parallel \tau_I(y)$	$TAM1$
$\tau_I(x \mathbin{⌊\!⌊} y) = \tau_I(x) \mathbin{⌊\!⌊} \tau_I(y)$	$TAM2$

Table 3.4.11: The axioms for abstraction in process algebra with multi-actions.

on Page 43, 2.4.48 on Page 53, and 3.4.2 on Page 67, where in all tables p, p', q, and q' range over $\mathcal{C}(\text{PA}^{\tau *}_{\delta\lambda}+\text{RN})$.

Definition 3.4.12. (Operational semantics) The operational semantics $\boldsymbol{\mathcal{S}}(\text{PA}^{\tau *}_{\delta\lambda}+\text{RN})$ is the process space $(\mathcal{C}(\text{PA}^{\tau *}_{\delta\lambda}+\text{RN}) \cup \{\surd\}, \boldsymbol{A}, \longrightarrow, \{\surd\})$ with rooted branching bisimilarity as the equivalence relation.

Property 3.4.13. (Congruence) *Rooted branching bisimilarity, $\sim_{rb}$, is a congruence for the operators of* $\text{PA}^{\tau *}_{\delta\lambda}+\text{RN}$.

Proof. All operators of $\text{PA}^{\tau *}_{\delta\lambda}+\text{RN}$ except the causal state operator, are treated as in the proofs of Properties 2.4.42 on Page 50 and 2.4.51 on Page 53. Let $\mathcal{R}$ be a rooted branching bisimulation such that $p\mathcal{R}q$, where p and q are closed $\text{PA}^{\tau *}_{\delta\lambda}+\text{RN}$ terms. Let $\lambda^I_s(\surd)$ be equal to $\surd$. It is straightforward to verify that $\{(\lambda^I_s(p), \lambda^I_s(q)) \mid p\mathcal{R}q \wedge I \in \mathcal{P}(U) \wedge s \in \mathcal{B}(U)\}$ is a rooted branching bisimulation between $\lambda^I_s(p)$ and $\lambda^I_s(q)$. □

Theorem 3.4.14. (Soundness) *For any closed terms $p, q \in \mathcal{C}(\text{PA}^{\tau *}_{\delta\lambda}+\text{RN})$,*
$\text{PA}^{\tau *}_{\delta\lambda}+\text{RN} \vdash p = q \Rightarrow \boldsymbol{\mathcal{S}}(\text{PA}^{\tau *}_{\delta\lambda}+\text{RN}) \models p = q.$

Proof. It is not difficult to construct a rooted branching bisimulation for each axiom of $\text{PA}^{\tau *}_{\delta\lambda}+\text{RN}$. □

Standard concurrency, expansion, fairness, and recursive equations. The extension of the theory $\text{PA}^{\tau *}_{\delta\lambda}+\text{RN}$ with any of these features is the same as in Section 2.4.5.

An algebraic semantics for P/T-net components. The algebraic black-box semantics for P/T-net components is defined using the definition of the algebraic glass-box semantics for net components and the abstraction operator of $\text{PA}^{\tau *}_{\delta\lambda}+\text{RN}$. The idea is simply to hide consumptions and productions that are internal to a component.

Definition 3.4.15. (Algebraic black-box semantics) Let $C = (N, I, s)$ be a P/T-net component in $\boldsymbol{\mathcal{C}}$. The algebraic semantics for the black-box behavior of C, denoted $\mathcal{C}^b[\![C]\!]$, is defined as follows:

$\mathcal{C}^b[\![C]\!] = \tau_I(\mathcal{C}^g[\![C]\!])$,
where $\mathcal{C}^g[\![C]\!]$ is the algebraic glass-box semantics of C as defined in Definition 3.4.6 on Page 67.

Example 3.4.16. It is important to note that the algebraic black-box semantics cannot be defined directly in terms of the action function α^b, similar to the definition of the algebraic glass-box semantics in terms of the action function α^g (Definition 3.4.6 on Page 67). Let C be again the P/T-net component of Figure 3.3.1 on Page 62. Let I be the set of internal places of C, $\{rdy, bsy, tcmd, tprdy, empty, pmt\}$. Assume the black-box behavior of C had been defined as follows.

$$\lambda^I_{[rdy,empty]}(\alpha^b(rcmd) * \delta \parallel \alpha^b(pmat) * \delta \parallel \alpha^b(sprdy) * \delta \parallel \alpha^b(omat) * \delta).$$

It follows from Definition 3.3.9 (Action function α^b) on Page 64 that this expression is equal to

$$\lambda^I_{[rdy,empty]}(([cmd], \mathbf{0}) * \delta \parallel ([imt^3], \mathbf{0}) * \delta \parallel (\mathbf{0}, [omt]) * \delta \parallel (\mathbf{0}, [prdy]) * \delta).$$

It is not difficult to see that, in the operational semantics $\mathcal{S}(\mathrm{PA}^{\tau*}_{\delta\lambda}+\mathrm{RN})$ given in Definition 3.4.12 on the facing page, this term can perform any of the four actions $\alpha^b(rcmd)$, $\alpha^b(pmat)$, $\alpha^b(sprdy)$, and $\alpha^b(omat)$. According to the operational black-box semantics $\mathcal{S}^b(\mathcal{C})$ for P/T-net components given in Definition 3.3.13 on Page 65, P/T-net component C can only fire transition *rcmd*. This means that the component can only perform action $\alpha^b(rcmd)$. Clearly, this discrepancy is undesirable. The reason for the discrepancy is that, in the algebraic semantics given in this example, the internal consumptions and productions of the component are hidden, by means of the black-box action function α^b, *before* the causal state operator is applied. Thus, the causal state operator misses crucial information about the causal relationships between the actions of the net-component. In Definition 3.4.15 on the facing page, first the glass-box behavior of a component is determined and then the internal consumptions and productions are hidden. Effectively, this means that the causal state operator is applied before the abstraction takes place.

Theorem 3.4.17. *Assume that the semantics for P/T-net components is the operational black-box semantics $\mathcal{S}^b(\mathcal{C})$ of Definition 3.3.13 on Page 65. For any P/T-net components (N, I, s) and (N, I, s') in $\mathcal{C}$, closed term p in $C(\mathrm{PA}^{\tau*}_{\delta\lambda}+\mathrm{RN})$, and action $a \in \mathbf{A}$,*

i) $(N, I, s)\ [a\rangle\ (N, I, s') \Rightarrow \mathcal{C}^b[\![(N, I, s)]\!] \xrightarrow{a} \mathcal{C}^b[\![(N, I, s')]\!]$,

ii) $\mathcal{C}^b[\![(N, I, s)]\!] \xrightarrow{a} p \Rightarrow$
$(\exists\, s' : s' \in \mathcal{B}(I) : p \equiv \mathcal{C}^b[\![(N, I, s')]\!] \wedge (N, I, s)\ [a\rangle\ (N, I, s'))$,

iii) $\mathcal{C}^b[\![(N, I, s)]\!] \not\xrightarrow{a} \surd$.

Proof. The proof uses Theorem 3.4.8 on Page 68, where a similar result is proven for the glass-box semantics for P/T-net components. Let N be equal to (P, T, F, W).

i) Assume $(N, I, s)\ [a\rangle\ (N, I, s')$. It follows from the definitions of the operational black-box semantics $\mathcal{S}^b(\mathcal{C})$ for P/T-net components (Definition 3.3.13 on Page 65) and the firing rule (Definition 3.3.8 on Page 64) that there must be a transition t in T, enabled in marking s, such that $a = \alpha^b(t)$ and $s' = s - \boldsymbol{iit} \uplus \boldsymbol{iot}$. The definition of the firing rule is such that the action function does not influence the marking s'. As a consequence, in the glass-box semantics $\mathcal{S}^g(\mathcal{C})$ of Definition 3.3.10 on Page 64, $(N, I, s)\ [\alpha^g(t)\rangle\ (N, I, s')$. Hence, Theorem 3.4.8 *i*) on Page 68 yields that

$\mathcal{C}^g[\![(N, I, s)]\!] \xrightarrow{\alpha^g(t)} \mathcal{C}^g[\![(N, I, s')]\!]$.

Definition 3.3.9 (Action functions α^g and α^b) on Page 64 yields that $a = \alpha^b(t) = \boldsymbol{f}_I(\alpha^g(t))$, where function $\boldsymbol{f}_I$ is the renaming function used to define the abstraction operator τ_I on Page 71. Hence, it follows from the derivation rules for the renaming function given in Table 2.4.24 on Page 43 that

$\tau_I(\mathcal{C}^g[\![(N, I, s)]\!]) \xrightarrow{a} \tau_I(\mathcal{C}^g[\![(N, I, s')]\!])$.

Definition 3.4.15 (Algebraic black-box semantics) on Page 72 completes the first part of the proof.

ii) Assume $\mathcal{C}^b[\![(N, I, s)]\!] \xrightarrow{a} p$. Definition 3.4.15 (Algebraic semantics) yields that

$\tau_I(\mathcal{C}^g[\![(N, I, s)]\!]) \xrightarrow{a} p$.

Let $\boldsymbol{f}_I$ be the renaming function used to define the abstraction operator τ_I on Page 71. According to the derivation rules for the renaming function given in Table 2.4.24 on Page 43, there must be a closed term $q \in \mathcal{C}(\mathrm{PA}^{\tau *}_{\delta\lambda}+\mathrm{RN})$ and an action $b \in \boldsymbol{A}$ with $\boldsymbol{f}_I(b) = a$ such that,

$p \equiv \tau_I(q)$,

and

$\mathcal{C}^g[\![(N, I, s)]\!]) \xrightarrow{b} q$.

It follows from Theorem 3.4.8 *ii*) on Page 68 that there exists a marking s' such that

$q \equiv \mathcal{C}^g[\![(N, I, s')]\!]$

and such that, in the glass-box semantics $\mathcal{S}^g(\mathcal{C})$ of Definition 3.3.10 on Page 64,

$(N, I, s)\ [b\rangle\ (N, I, s')$.

Hence, Definition 3.3.8 (Firing rule) also on Page 64 yields that there must be a transition $t \in T$, enabled in marking s, such that $b = \alpha^g(t)$ and $s' = s - \boldsymbol{i}\boldsymbol{i}t \uplus \boldsymbol{i}\boldsymbol{o}t$. Definition 3.3.9 (Action functions α^g and α^b) on Page 64 yields that $a = \boldsymbol{f}_I(b) = \boldsymbol{f}_I(\alpha^g(t)) = \alpha^b(t)$. Since the choice for the action function does not affect the consumption and production of tokens when firing a transition, it follows that, in the black-box semantics $\mathcal{S}^b(\mathcal{C})$ of Definition 3.3.13 on Page 65,

$(N, I, s)\ [a\rangle\ (N, I, s')$.

Since $p \equiv \tau_I(q)$ and $q \equiv \mathcal{C}^g[\![(N, I, s')]\!]$, Definition 3.4.15 (Algebraic black-box semantics) on Page 72 yields,

$p \equiv \mathcal{C}^b[\![(N, I, s')]\!]$,

which completes the second part of the proof.

iii) Assume $\mathcal{C}^b[\![(N, I, s)]\!] \xrightarrow{a} \surd$. It follows from Definition 3.4.15 (Algebraic semantics) that

$\tau_I(\mathcal{C}^g[\![(N, I, s)]\!]) \xrightarrow{a} \surd$.

It follows from the derivation rules for the renaming operator in Table 2.4.24 on Page 43 that there is an action $b \in \boldsymbol{A}$ with $\boldsymbol{f}_I(b) = a$ such that,

$\mathcal{C}^g[\![(N, I, s)]\!] \xrightarrow{b} \surd$.

Clearly this contradicts Theorem 3.4.8 on Page 68. Hence,
$\mathcal{C}^b[\![(N, I, s)]\!] \overset{a}{\not\longrightarrow} \surd$,
which completes the proof.

□

Example 3.4.18. Consider again the P/T-net component C of Figure 3.3.1 on Page 62. As before, the identifier t of a transition of C is an abbreviation of the action $\alpha^g(t)$ (see Definition 3.3.9 on Page 64). Recall that I is the set of internal places of C, namely $\{rdy, bsy, tcmd, tprdy, empty, pmt\}$.

$$
\begin{array}{ll}
 & \mathcal{C}^b[\![C]\!] \\
= & \quad \{ \text{ Definition 3.4.15 (Algebraic black-box semantics) on Page 72 } \} \\
 & \tau_I(\mathcal{C}^g[\![C]\!]) \\
= & \quad \{ \text{ Result of Example 3.4.10 on Page 70 } \} \\
 & \tau_I((rcmd \cdot pmat \cdot omat \cdot sprdy) * \delta) \\
= & \quad \{ \; TA5, TA4 \text{ (repeatedly) } \} \\
 & (\tau_I(rcmd) \cdot \tau_I(pmat) \cdot \tau_I(omat) \cdot \tau_I(sprdy)) * \tau_I(\delta) \\
= & \quad \{ \; TA2 \text{ (repeatedly), } TA1 \; \} \\
 & (([cmd], \mathbf{0}) \cdot ([imt^3], \mathbf{0}) \cdot (\mathbf{0}, [omt]) \cdot (\mathbf{0}, [prdy])) * \delta
\end{array}
$$

It is not difficult to see that the operational semantics for $\mathcal{C}^b[\![C]\!]$ is identical to the operational black-box semantics for C given in Example 3.3.14 on Page 65. Clearly, Theorem 3.4.17 on Page 73 is satisfied.

3.5 Modular P/T Nets

In the previous two sections, P/T-net components, their operational semantics, and an algebraic characterization of their semantics have been presented. As already mentioned, P/T-net components form the basis for modular P/T nets. Modular P/T nets are a simple form of hierarchical P/T nets. A modular P/T net is simply a collection of P/T-net components, also called subnets, that are connected via places. This means that a modular P/T net consists of only two hierarchical levels. The reason for this restriction is simplicity. It is much harder to reason formally about hierarchical P/T nets with an arbitrary number of hierarchical levels than it is to reason about modular P/T nets. The basic ideas and important results of this chapter, on the other hand, can be easily explained by means of modular P/T nets. The modularity construct of modular P/T nets is essentially the same as the hierarchy construct in the hierarchical Petri nets of (VAN HEE, 1994). It is also one of the constructs used in (JENSEN, 1992). Furthermore, it is supported by tools as Design/CPN and ExSpect.

The interface function *ifc* on P/T-net components is generalized to sets of components by taking the union of the interfaces of all the components in a set.

Definition 3.5.1. (Modular P/T net) A (marked) modular P/T net is a pair (M, s), with M in turn a tuple (S, P, I, f), where *i*) $S \subseteq \mathcal{C}$ is a non-empty set of marked P/T-net components which are also called the *subnets* of M;

ii) $P \subseteq U$ is the set of *places* of M;

iii) $I \subseteq P$ is the set of *internal places* of M;

iv) $f : ifc(S) \to P$ is a surjective function, called the *place-fusion function*, mapping the pins of the subnets of M to places in P;

v) $s \in \mathcal{B}(I)$ is the internal *marking* of M.

The set $P \backslash I$ is the set of pins or the interface of M, denoted $ifc(M)$. A modular P/T net must satisfy the following restrictions.

i) The set of places of any component in S and the set of places P of the modular net are disjoint;

ii) The sets of places of any two different components in S are disjoint;

iii) The sets of transitions of two different components in S are disjoint.

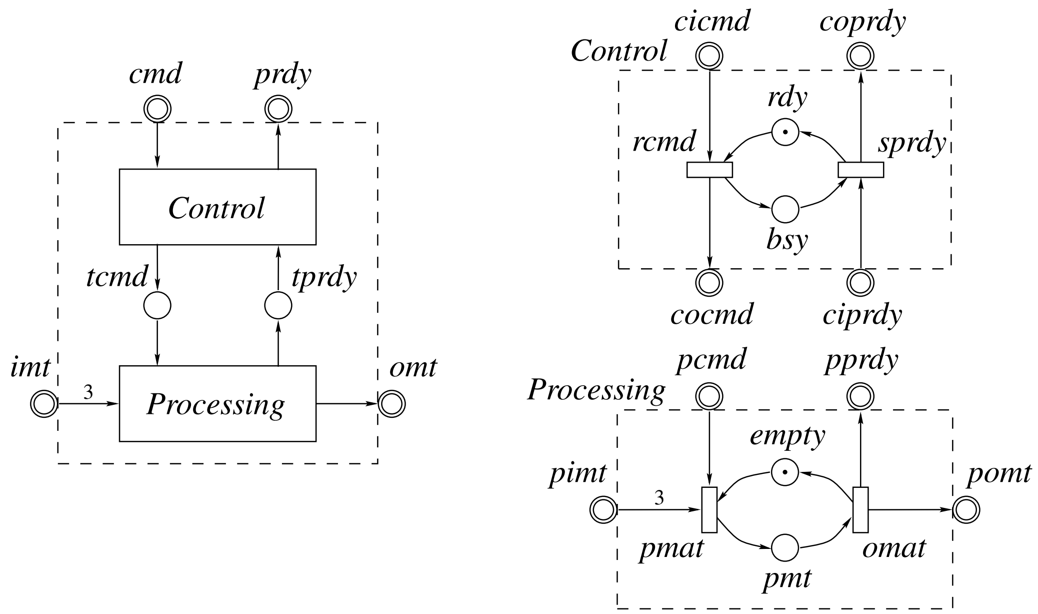


Figure 3.5.2: An example of a modular P/T net.

Example 3.5.3. Figure 3.5.2 shows an example of a modular P/T net. The top level is shown in the left half of the figure. The modular net contains two components, called *Control* and *Processing*. The right half of Figure 3.5.2 shows these two components. The place-fusion function f is defined as follows:

$cicmd \mapsto cmd$	$pimt \mapsto imt$
$cocmd \mapsto tcmd$	$pcmd \mapsto tcmd$
$ciprdy \mapsto tprdy$	$pprdy \mapsto tprdy$
$coprdy \mapsto prdy$	$pomt \mapsto omt$

Intuitively, the modular P/T net of Figure 3.5.2 models the same production unit as the one in Figure 3.3.1 on Page 62. The only difference is that, in the modular net, the production unit is divided into a control part and a processing part. The two parts exchange information via places. The interface of the modular net is identical to the interface of the component in Figure 3.3.1.

The next step is to define operational semantics for modular P/T nets. As usual, the basis for any operational semantics is a firing rule. It is possible to define a firing rule directly on modular P/T nets using the firing rule for P/T-net components. This yields an elegant, compositional definition which is useful when defining algebraic semantics for modular P/T nets. It is also possible to define a firing rule for modular P/T nets in an indirect way. The hierarchy in a modular net can be removed by *unfolding* the modular net into a net component. The firing rule of a modular net is then defined as the firing rule of its unfolding. The first approach yields the most elegant definition of the firing rule. The second one guarantees consistency with the semantics for P/T-net components. Therefore, in the remainder of this section, the following approach is taken. First, a compositional definition of the firing rule for modular P/T nets is given. Based on this firing rule, three different operational semantics are defined, called the glass-box semantics, the system semantics, and the black-box semantics. The system semantics defines an interesting view on modular nets that lies in between the glass-box view and the black-box view. Second, the unfolding of a modular P/T net is defined. Third, it is shown that the glass-box and black-box semantics of a modular P/T net correspond to the glass-box and black-box semantics of its unfolding. As explained below, it is also possible to define a semantics for P/T-net components that corresponds to the system semantics for modular nets. However, for P/T-net components, such a semantics is not meaningful, since it uses the hierarchical structure of modular nets, which is lost when unfolding a modular net into a component.

A renaming function $f : U \to U$ is generalized to bags over U as follows. For any bag X over U, the value $f(X)$ is defined as $(\uplus x : x \in X : [f(x)^{X(x)}])$. A transition in some component of a modular P/T net is enabled if two conditions are satisfied. First, it must be enabled in the component. Second, the marking of the modular net must contain sufficient tokens in the internal places connected to the input pins of the transition. The second requirement is formalized as follows. Let (M, s), with $M = (S, P, I, f)$, be a modular P/T net; let t be a transition in some component C in S. The bag of tokens that t needs in internal places of the modular net in order to be enabled is equal to $f(\boldsymbol{oi}t) \upharpoonright I$. This means that the bag of tokens required in input pins of t is renamed to a bag of tokens in places in P and, subsequently, restricted to internal places of the modular net.

Definition 3.5.4. (Transition enabling) Let (M, s), with $M = (S, P, I, f)$, be a modular P/T net. Let t be a transition in some component C in S. Transition t is enabled, denoted $(M, s)[t\rangle$, if and only if $C[t\rangle$ (see Definition 3.3.5 on Page 63) and $f(\boldsymbol{oi}t) \upharpoonright I \leq s$.

The set of all modular P/T nets is denoted $\mathcal{M}$.

Definition 3.5.5. (Firing rule) The firing rule $\_\,[\_\rangle\,\_ \subseteq \mathcal{M} \times A \times \mathcal{M}$ is the smallest relation satisfying for any modular P/T net (M, s) with $M = (S, P, I, f)$, component C

in S with set of transitions T and action function $\alpha : T \to \boldsymbol{A}$, and transition t in T,

$(M, s)[t\rangle \Rightarrow (M, s)\ [\alpha(t)\rangle\ (M', s - f(\boldsymbol{oit}) \restriction I \uplus f(\boldsymbol{oot}) \restriction I)$,

where M' is identical to M except that subnet C is replaced by the subnet C' obtained by firing transition t in C (see Definition 3.3.8 on Page 64).

Action functions determine what action a modular net performs when firing a transition. As mentioned, three different operational semantics can be defined for modular P/T nets. In the first one, all consumptions and productions of the modular net including those inside components are visible. The modular net is a glass box. In the second semantics, the consumptions and productions inside the components of a modular net are hidden, but all other consumptions and productions of are visible. This semantics is useful when one is interested in the behavior of the modular net without seeing all behavioral details of its components. This view is called the *system view* of the modular net. The idea is that the modular P/T net models the system under development; its components are standard building blocks whose internal behavior is of no interest to the developer. In the final semantics, all internal consumptions and productions are hidden. The modular net behaves as a black box. This semantics is useful when the modular net itself is meant to be a building block in a larger system model.

Let $f : U \to U$ be some renaming function. If f is a partial function with domain D, then $\bar{f} : U \to U$ is a total function defined as follows: For any $d \in D$, $\bar{f}(d) = f(d)$; for any $u \in U \backslash D$, $\bar{f}(u) = u$. Let (M, s), with $M = (S, P, I, f)$, be a modular P/T net. In the definition of action functions for modular P/T nets, the place-fusion function f and its extension $\bar{f}$ are used to rename consumptions of tokens from and productions of tokens into pins of components in such a way that they appear as consumptions from and productions into places in P. Let C be a component in S and t a transition in C. When firing transition t, its consumption from pins of C is equal to $\boldsymbol{oit}$. As already mentioned, this translates to the consumption of the bag of tokens $f(\boldsymbol{oit})$ from places in P. The total consumption of t from places in C is equal to $\boldsymbol{it}$. The renaming function $\bar{f}$ renames consumptions from pins of C to consumptions from places in P, but it does not affect consumptions that are internal to component C: $\bar{f}(\boldsymbol{it}) = \bar{f}(\boldsymbol{oit} \uplus \boldsymbol{iit}) = \bar{f}(\boldsymbol{oit}) \uplus \bar{f}(\boldsymbol{iit}) = f(\boldsymbol{oit}) \uplus \boldsymbol{iit}$.

Definition 3.5.6. (Action functions) Let (M, s), with $M = (S, P, I, f)$, be a modular P/T net. Let C be a component in S with set of transitions T.

Glass-box action function: For any $t \in T$, $\mu^g(t) = (\bar{f}(\boldsymbol{it}), \bar{f}(\boldsymbol{ot}))$;

System-view action function: For any $t \in T$, $\mu^s(t) = (f(\boldsymbol{oit}), f(\boldsymbol{oot}))$;

Black-box action function: For any $t \in T$, $\mu^b(t) = (f(\boldsymbol{oit}) \restriction ifc(M), f(\boldsymbol{oot}) \restriction ifc(M))$.

Definition 3.5.7. (Operational glass-box semantics) The glass-box behavior of modular P/T nets is formalized by the operational semantics $\mathcal{S}^g(\mathcal{M})$. It consists of the process space $(\mathcal{M}, \boldsymbol{A}, [\ \rangle, \emptyset)$, where the firing rule is instantiated with the glass-box action function μ^g. The equivalence on processes is strong bisimilarity.

Definition 3.5.8. (Operational system semantics) The system behavior of modular P/T nets is formalized by the operational semantics $\mathcal{S}^s(\mathcal{M})$. It consists of the process space

$(\mathcal{M}, A, [\,\rangle\,, \emptyset)$, where the firing rule is instantiated with the system-view action function μ^s. The equivalence on processes is rooted branching bisimilarity.

Definition 3.5.9. (Operational black-box semantics) The black-box behavior of modular P/T nets is formalized by the operational semantics $\mathcal{S}^b(\mathcal{M})$. It consists of the process space $(\mathcal{M}, A, [\,\rangle\,, \emptyset)$, where the firing rule is instantiated with the black-box action function μ^b. The equivalence on processes is rooted branching bisimilarity.

$$\begin{tikzcd}
 & s_0, c_0, p_0 \arrow[dl, "rcmd"'] & \\
s_1, c_1, p_1 \arrow[dr, "pmat"'] & & s_3, c_3, p_3 \arrow[ul, "sprdy"'] \\
 & s_2, c_2, p_2 \arrow[ur, "omat"'] &
\end{tikzcd}$$

$s_0 = \mathbf{0}$	$c_0 = [rdy]$	$p_0 = [empty]$
$s_1 = [tcmd]$	$c_1 = [bsy]$	$p_1 = [empty]$
$s_2 = \mathbf{0}$	$c_2 = [bsy]$	$p_2 = [pmt]$
$s_3 = [tprdy]$	$c_3 = [bsy]$	$p_3 = [empty]$

Figure 3.5.10: The operational semantics of the modular production unit.

Example 3.5.11. Consider the modular net of Figure 3.5.2 on Page 76. Figure 3.5.10 shows its operational semantics. The firing rule is depicted by arrows. Processes are represented by the marking of the places of the modular net and the markings of its subnets. For any i, with $0 \leq i \leq 3$, s_i is the marking of the places *tcmd* and *tprdy*, c_i is the marking of component *Control*, and p_i is the marking of *Processing*. Transition identifiers are used as an abbreviation of the corresponding action. Depending on what action function is chosen, Figure 3.5.10 shows the glass-box behavior, the system behavior, or the black-box behavior of the net of Figure 3.5.2.

To show that the above semantics are consistent with the semantics for P/T-net components, the unfolding of a modular P/T net is defined. The unfolding removes the hierarchy from a modular P/T net resulting in a P/T-net component. For any P/T-net component $C = ((P, T, F, W), I, s)$, the notation $C.*$ is used to denote element $*$ of the component, where $* \in \{P, T, F, W, I, s\}$.

Definition 3.5.12. (Unfolding) The unfolding of a modular P/T net (M, s) in $\mathcal{M}$, where $M = (S, P, I, f)$, is a P/T-net component $\mathcal{U}(M, s) = ((P_u, T_u, F_u, W_u), I_u, s_u)$ defined as follows:

i) The set of places of the unfolding contains all the places of the components, where pins of components are renamed according to the place-fusion function of M:
$P_u = \bar{f}(\cup C : C \in S : C.P)$.
Note that P is a subset of P_u, because the place-fusion function is surjective.

ii) The set of transitions contains all the transitions of all the components:
$T_u = (\cup C : C \in S : C.T)$.

iii) The flow relation of the unfolding of M contains all the arcs of all the components where arcs connected to a pin of a component are connected to the place to which the pin is mapped by the place-fusion function:
$F_u = (\cup C : C \in S : \{(\bar{f}(p), t) \mid pC.Ft\} \cup \{(t, \bar{f}(p)) \mid tC.Fp\})$.

Note that several arcs of a component may collapse into a single arc in the unfolding when multiple pins of the component are mapped to the same place in P. It is not possible that arcs from two different components result in a single arc in the unfolding.

iv) The weight of an arc in the unfolding is simply the sum of the weights of all arcs in a component resulting in this one arc in the unfolding. For any $(p, t) \in F_u$, where t is a transition in some component $C \in S$,
$W_u(p, t) = (+\, p' :\ p'C.Ft \wedge \bar{f}(p') = p :\ C.W(p', t))$.
For any $(t, p) \in F_u$, where t is a transition in some component $C \in S$,
$W_u(t, p) = (+\, p' :\ tC.Fp' \wedge \bar{f}(p') = p :\ C.W(t, p'))$.

v) The set of internal places of the unfolding contains the internal places of M and, in addition, all internal places of the components:
$I_u = I \cup (\cup\, C :\ C \in S :\ C.I)$.

vi) The marking is simply the marking of M plus all the markings of the components:
$s_u = s \uplus (\uplus\, C :\ C \in S :\ C.s)$.

Example 3.5.13. The unfolding of the modular P/T net of Figure 3.5.2 on Page 76 is the P/T-net component of Figure 3.3.1 on Page 62. This validates the claim made earlier that the modular net of Figure 3.5.2 and the component of Figure 3.3.1 model the same system.

As mentioned, it is desirable that a modular P/T net and its unfolding have the same operational semantics. The correspondence should hold for the glass-box and black-box semantics. Technically, it is also possible to define the system semantics of an unfolded net, provided that the hierarchical structure of the original modular net is remembered in one way or the other. However, such a semantics is not very meaningful for P/T-net components. Arbitrary P/T-net components simply do not have a modular structure that can be used as a basis for such a semantics.

Theorem 3.5.14. *Assume that* $\mathcal{S}^*(\mathcal{C})$ *and* $\mathcal{S}^*(\mathcal{M})$, *with* $* \in \{g, b\}$, *are the operational semantics for P/T-net components and modular P/T nets respectively. Let* (M, s) *with* $M = (S, P, I, f)$ *and* (M', s') *be modular P/T nets in* $\mathcal{M}$. *For any component* $V \in \mathcal{C}$ *and action* $a \in \boldsymbol{A}$,

i) $(M, s)\ [a\rangle\ (M', s') \Rightarrow \mathcal{U}(M, s)\ [a\rangle\ \mathcal{U}(M', s')$,

ii) $\mathcal{U}(M, s)\ [a\rangle\ V \Rightarrow$
$(\exists\, (M', s') :\ (M', s') \in \mathcal{M} :\ V = \mathcal{U}(M', s') \wedge (M, s)\ [a\rangle\ (M', s'))$.

Proof. The theorem is proven under the assumption that the glass-box semantics is chosen for modular nets and net components. The proof for the black-box semantics is identical except for a few minor details in the proof of the first auxiliary property given below.

Let the unfolding $\mathcal{U}(M, s)$ be the tuple $((P_u, T_u, F_u, W_u), I_u, s_u)$. It is clear that the components of a modular P/T net and its unfolding contain the same transitions. To distinguish functions $\boldsymbol{i}$, $\boldsymbol{o}$, $\boldsymbol{ii}$, $\boldsymbol{oi}$, $\boldsymbol{io}$, and $\boldsymbol{oo}$ in a component C of M from those in the unfolding $\mathcal{U}(M, s)$, the latter are subscripted with the character "u."

The following two auxiliary results are useful in the remainder of the proof. Let t be some transition in some component C of M.

(a) The glass-box action function for the components of a modular net and the glass-box action function for its unfolding coincide on t:

$\mu^g(t) = \alpha^g(t)$.

The proof is straightforward. Definition 3.5.12 (Unfolding) on Page 79 yields that $\boldsymbol{i}_u t = \bar{f}(\boldsymbol{i}t)$ and $\boldsymbol{o}_u t = \bar{f}(\boldsymbol{o}t)$. That is, the input (output) places of t in the unfolding are the input (output) places of t in component C renamed according to the place-fusion function. Definitions 3.5.6 (Action function μ^g) on Page 78 and 3.3.9 (Action function α^g) on Page 64 yield that $\mu^g(t) = (\bar{f}(\boldsymbol{i}t), \bar{f}(\boldsymbol{o}t)) = (\boldsymbol{i}_u t, \boldsymbol{o}_u t) = \alpha^g(t)$.

(b) Transition t is enabled in the modular net if and only if it is enabled in its unfolding:

$(M, s)[t\rangle \;\Leftrightarrow\; \mathcal{U}(M, s)[t\rangle$.

Definitions 3.5.4 (Transition enabling in modular P/T nets) on Page 77 and 3.3.5 (Transition enabling in P/T-net components) on Page 63 yield that the following must be shown:

$f(\boldsymbol{oi}t) \restriction I \le s \wedge \boldsymbol{ii}t \le C.s \;\Leftrightarrow\; \boldsymbol{ii}_u t \le s_u$.

The derivation below is a straightforward result of Definitions 3.3.2 (P/T-net component) on Page 62, 3.5.1 (modular P/T net) on Page 75, and 3.5.12 (Unfolding) on Page 79. It states that the internal input places of t in the unfolding are equal to the observable input places of t in C renamed according to the place-fusion function and restricted to the internal places of the modular net M plus the internal input places of t in C.

$$\begin{aligned}\boldsymbol{ii}_u t &= \boldsymbol{i}_u t \restriction I_u \\ &= \bar{f}(\boldsymbol{i}t) \restriction (I \cup (\cup D : D \in S : D.I)) \\ &= (f(\boldsymbol{oi}t) \uplus \boldsymbol{ii}t) \restriction (I \cup (\cup D : D \in S : D.I)) \\ &= f(\boldsymbol{oi}t) \restriction I \uplus \boldsymbol{ii}t.\end{aligned}$$

Definition 3.5.12 (Unfolding) on Page 79 yields that $s_u = s \uplus (\uplus D : D \in S : D.s)$. Hence, the proof obligation reduces to:

$f(\boldsymbol{oi}t) \restriction I \le s \wedge \boldsymbol{ii}t \le C.s \;\Leftrightarrow\; f(\boldsymbol{oi}t) \restriction I \uplus \boldsymbol{ii}t \le s \uplus (\uplus D : D \in S : D.s)$.

The implication from left to right is clearly true. The converse implication follows easily from the following two observations. First, $f(\boldsymbol{oi}t) \restriction I$ and s are both bags over I, whereas $\boldsymbol{ii}t$, $C.s$, and $(\uplus D : D \in S : D.s)$ do not contain any elements in I. Second, since $\boldsymbol{ii}t$ only contains elements of $C.I$, it follows that $\boldsymbol{ii}t \le (\uplus D : D \in S : D.s)$ implies $\boldsymbol{ii}t \le C.s$.

Using these two auxiliary results, the proof of the main theorem is as follows.

i) Assume $(M, s)\ [a\rangle\ (M', s')$. It must be shown that $\mathcal{U}(M, s)\ [a\rangle\ \mathcal{U}(M', s')$. Definition 3.5.5 (Firing rule) on Page 77 implies that there is a transition t in some component $C \in S$, such that $a = \mu^g(t)$, $(M, s)[t\rangle$, and $s' = s - f(\boldsymbol{oi}t) \restriction I \uplus f(\boldsymbol{oo}t) \restriction I$. Furthermore, tuple M' is equal to (S', P, I, f), where S' is S except that C is replaced by component C' obtained from C by firing transition t (see Definition 3.3.8 (Firing rule) on Page 64). The two auxiliary results above imply that $\mathcal{U}(M, s)[t\rangle$ and that $\alpha^g(t) = a$. It follows from the definitions of the firing rules for modular P/T nets and P/T net components (Definitions 3.5.5 and 3.3.8) and the definition of the unfolding of modular nets (Definition 3.5.12 on Page 79) that $\mathcal{U}(M, s)$

and $\mathcal{U}(M', s')$ only differ in their marking. This means that $\mathcal{U}(M', s')$ is the tuple $((P_u, T_u, F_u, W_u), I_u, s'_u)$, where $s'_u = s' \uplus (\uplus D : D \in S' : D.s)$. Hence, according to Definition 3.3.8 (Firing rule for P/T-net components) on Page 64, it suffices to show that $s'_u = s_u - \boldsymbol{ii}_u t \uplus \boldsymbol{io}_u t$. The proof is as follows. Recall that C' is the P/T-net component obtained from C by firing transition t. Hence, $C'.s = C.s - \boldsymbol{ii}t \uplus \boldsymbol{io}t$. Recall from above that $\boldsymbol{ii}_u t = f(\boldsymbol{oi}t) \lceil I \uplus \boldsymbol{ii}t$. There is a similar result for the bag of internal output places of t in the unfolding: $\boldsymbol{io}_u t = f(\boldsymbol{oo}t) \lceil I \uplus \boldsymbol{io}t$. The following derivation is a straightforward result of Definitions 3.3.2 (P/T-net component) on Page 62, 3.5.1 (modular P/T net) on Page 75, 3.5.12 (Unfolding) on Page 79, and 3.5.5 (Firing rule) on Page 77.

$$\begin{aligned} s'_u &= s' \uplus (\uplus D : D \in S' : D.s) \\ &= (s - f(\boldsymbol{oi}t) \lceil I \uplus f(\boldsymbol{oo}t) \lceil I) \uplus \\ &\qquad (\uplus D : D \in S \wedge D \neq C : D.s) \uplus (C.s - \boldsymbol{ii}t \uplus \boldsymbol{io}t) \\ &= s \uplus (\uplus D : D \in S : D.s) - (f(\boldsymbol{oi}t) \lceil I \uplus \boldsymbol{ii}t) \uplus (f(\boldsymbol{oo}t) \lceil I \uplus \boldsymbol{io}t) \\ &= s_u - \boldsymbol{ii}_u t \uplus \boldsymbol{io}_u t. \end{aligned}$$

This completes the first part of the proof.

ii) Assume $\mathcal{U}(M, s)\ [a\rangle\ V$. It must be shown that there is a modular net (M', s') such that $V = \mathcal{U}(M', s')$ and $(M, s)\ [a\rangle\ (M', s')$. It follows from the definition of the firing rule for net components (Definition 3.3.8 on Page 64) and the assumption that $\mathcal{U}(M, s)$ is the tuple $((P_u, T_u, F_u, W_u), I_u, s_u)$ that there is a transition t in $\mathcal{U}(M, s)$ such that $a = \alpha^g(t)$, $\mathcal{U}(M, s)[t\rangle$, and $V = ((P_u, T_u, F_u, W_u), I_u, s_u - \boldsymbol{ii}_u t \uplus \boldsymbol{io}_u t)$. Definition 3.5.12 (Unfolding) on Page 79 implies that t is a transition in some component C of M. It follows from the auxiliary results above that $\mu^g(t) = a$ and $(M, s)[t\rangle$. Let (M', s') be the modular net obtained by firing transition t. Hence, $s' = s - f(\boldsymbol{oi}t) \lceil I \uplus f(\boldsymbol{oo}t) \lceil I$ and $M' = (S', P, I, f)$, where S' is S with C replaced by C' which is obtained from C by firing transition t (see Definition 3.3.8 on Page 64). It suffices to show that $V = \mathcal{U}(M', s')$. Definitions 3.5.5 (Firing rule for modular nets) on Page 77 and 3.5.12 (Unfolding) on Page 79 yield that $\mathcal{U}(M', s')$ differs from $\mathcal{U}(M, s)$ only in its marking. Hence, $\mathcal{U}(M', s') = ((P_u, T_u, F_u, W_u), I_u, s'_u)$ with $s'_u = s' \uplus (\uplus D : D \in S' : D.s)$. The derivation in Part *i*) of the proof shows that $s'_u = s_u - \boldsymbol{ii}_u t \uplus \boldsymbol{io}_u t$. Hence, $V = \mathcal{U}(M', s')$, which completes the proof. □

Example 3.5.15. Clearly, Theorem 3.5.14 on Page 80 is satisfied for the modular P/T net of Figure 3.5.2 on Page 76 and its unfolding shown in Figure 3.3.1 on Page 62 (see Examples 3.3.12 and 3.3.14 on Page 65 and 3.5.11 on Page 79).

3.6 Algebraic Semantics for Modular P/T Nets

In order to support compositional design in terms of Petri nets, this section presents algebraic characterizations of the three operational semantics for modular P/T nets given in the previous section.

3.6.1 Glass-box semantics

The algebraic semantics for the glass-box behavior of modular P/T nets is defined in the process algebra $\mathrm{PA}^*_{\delta\lambda} + \mathrm{RN}$. The algebraic glass-box semantics is compositional in the sense that it is defined in terms of the algebraic glass-box semantics for P/T-net components (see Definition 3.4.6 on Page 67). The unrestricted behavior of a modular P/T net is the parallel composition of the behavior of all its components, where consumptions and productions are renamed according to the place-fusion function of the modular net. The state operator as introduced in Section 3.4 is used to restrict the behavior to all possible firing sequences.

Let (M, s) with $M = (S, P, I, f)$ be a modular P/T net in $\boldsymbol{\mathcal{M}}$. Using the place-fusion function f, a renaming function $\boldsymbol{f} : \boldsymbol{A} \cup \{\delta\} \to \boldsymbol{A} \cup \{\delta\}$ is defined as follows: $\boldsymbol{f}(\delta) = \delta$; for any $(c, p) \in \boldsymbol{A}, \boldsymbol{f}(c, p) = (\bar{f}(c), \bar{f}(p))$.

Definition 3.6.1. (Algebraic glass-box semantics) Let (M, s) with $M = (S, P, I, f)$ be a modular P/T net in $\boldsymbol{\mathcal{M}}$. The algebraic semantics for the glass-box behavior of (M, s), denoted $\mathcal{M}^g[\![(M, s)]\!]$, is defined as follows:

$\mathcal{M}^g[\![(M, s)]\!] = \lambda^I_s(\| C : C \in S : \rho_{\boldsymbol{f}}(\mathcal{C}^g[\![C]\!]))$.

Note that the superscript of the causal state operator only contains the internal places of the modular net itself and not the internal places of the components. It is not necessary that the state operator in the definition above affects consumptions and productions that are internal to a component, because the algebraic semantics of the component is already restricted to all possible transition firings.

The following theorem states that the algebraic semantics and the operational semantics for modular P/T nets coincide. Its proof is relatively straightforward because both the algebraic semantics and the firing rule for modular P/T nets are defined in a compositional way. As a result, it is possible to use Theorem 3.4.8 on Page 68 in the proof.

Theorem 3.6.2. *Assume that the semantics for modular P/T nets is the operational glass-box semantics $\boldsymbol{\mathcal{S}}^g(\boldsymbol{\mathcal{M}})$ of Definition 3.5.7 on Page 78. For any two modular P/T nets (M, s) with $M = (S, P, I, f)$ and (M', s') in $\boldsymbol{\mathcal{M}}$, closed term p in $\mathcal{C}(\mathrm{PA}^*_{\delta\lambda} + \mathrm{RN})$, and action $a \in \boldsymbol{A}$,*

i) $(M, s)\ [a\rangle\ (M', s') \Rightarrow \mathcal{M}^g[\![(M, s)]\!] \xrightarrow{a} \mathcal{M}^g[\![(M', s')]\!]$,

ii) $\mathcal{M}^g[\![(M, s)]\!] \xrightarrow{a} p \Rightarrow$
$(\exists\, (M', s') : (M', s') \in \boldsymbol{\mathcal{M}} : p \equiv \mathcal{M}^g[\![(M', s')]\!] \wedge (M, s)\ [a\rangle\ (M', s'))$,

iii) $\mathcal{M}^g[\![(M, s)]\!] \not\xrightarrow{a} \surd$.

Proof.

i) Assume $(M, s)\,[a\rangle\,(M', s')$. It must be shown that $\mathcal{M}^g[\![(M, s)]\!] \xrightarrow{a} \mathcal{M}^g[\![(M', s')]\!]$. It follows from the definition of the firing rule (Definition 3.5.5 on Page 77) that there is a transition t in some component C of M such that

(a) $a = \mu^g(t)$,

(b) $(M, s)[t\rangle$,

(c) $s' = s - f(\boldsymbol{oit}) \upharpoonright I \uplus f(\boldsymbol{oot}) \upharpoonright I$, and

(d) $M' = (S', P, I, f)$, where S' is equal to S with C replaced by C' which is obtained from C by firing transition t (Definition 3.3.8 on Page 64).

Definition 3.6.1 (Algebraic semantics) on the previous page implies that the following must be shown:

$$\lambda^I_s(\| D : D \in S : \rho_f(\mathcal{C}^g[\![D]\!])) \xrightarrow{a} \lambda^I_{s'}(\| D : D \in S' : \rho_f(\mathcal{C}^g[\![D]\!])).$$

It follows from Property (d) and Theorem 3.4.8 on Page 68 that

$$\mathcal{C}^g[\![C]\!] \xrightarrow{\alpha^g(t)} \mathcal{C}^g[\![C']\!].$$

Definitions 3.3.9 (Action function α^g) on Page 64, 3.5.6 (Action function μ^g) on Page 78, and the definition of the renaming function $\boldsymbol{f}$ above yield that $\boldsymbol{f}(\alpha^g(t)) = (\bar{f}(\boldsymbol{it}), \bar{f}(\boldsymbol{ot})) = \mu^g(t)$. Hence, it follows from Property (a) and the derivation rules in Table 2.4.24 on Page 43 that

$$\rho_f(\mathcal{C}^g[\![C]\!]) \xrightarrow{a} \rho_f(\mathcal{C}^g[\![C']\!]).$$

Since S' is equal to S except for components C and C', again the derivation rules in Table 2.4.24 on Page 43 yield that

$$(\| D : D \in S : \rho_f(\mathcal{C}^g[\![D]\!])) \xrightarrow{a} (\| D : D \in S' : \rho_f(\mathcal{C}^g[\![D]\!])).$$

Definition 3.5.6 (Action function μ^g) on Page 78 and Property (a) yield that $\boldsymbol{c}a \upharpoonright I = \boldsymbol{c}\mu^g(t) \upharpoonright I = \bar{f}(\boldsymbol{it}) \upharpoonright I = (f(\boldsymbol{oit}) \uplus \boldsymbol{iit}) \upharpoonright I = f(\boldsymbol{oit}) \upharpoonright I$. Similarly, $\boldsymbol{p}a \upharpoonright I = f(\boldsymbol{oot}) \upharpoonright I$. Hence, Property (c) implies that $s' = s - \boldsymbol{c}a \upharpoonright I \uplus \boldsymbol{p}a \upharpoonright I$. It follows from Property (b) and the definition of transition enabling (Definition 3.5.4 on Page 77) that $f(\boldsymbol{oit}) \upharpoonright I \leq s$, which means that $\boldsymbol{c}a \upharpoonright I \leq s$. Hence, the derivation rules for the causal state operator in Table 3.4.2 on Page 67 yield that

$$\lambda^I_s(\| D : D \in S : \rho_f(\mathcal{C}^g[\![D]\!])) \xrightarrow{a} \lambda^I_{s'}(\| D : D \in S' : \rho_f(\mathcal{C}^g[\![D]\!])),$$

which completes the first part of the proof.

ii) Assume $\mathcal{M}^g[\![(M, s)]\!] \xrightarrow{a} p$. It must be shown that there exists a modular P/T net (M', s') such that $p \equiv \mathcal{M}^g[\![(M', s')]\!]$ and $(M, s)\ [a\rangle\ (M', s')$. It follows from Definition 3.6.1 (Algebraic semantics) on the preceding page that

$$\lambda^I_s(\| D : D \in S : \rho_f(\mathcal{C}^g[\![D]\!])) \xrightarrow{a} p.$$

It follows from the derivation rules in Table 3.4.2 on Page 67 that there exists a closed term $q \in \mathcal{C}(\mathrm{PA}^*_{\delta\lambda}+\mathrm{RN})$ such that $\boldsymbol{c}a \upharpoonright I \leq s$,

$$p \equiv \lambda^I_{s'}(q),$$

with $s' = s - \boldsymbol{c}a \upharpoonright I \uplus \boldsymbol{p}a \upharpoonright I$, and

$$(\| D : D \in S : \rho_f(\mathcal{C}^g[\![D]\!])) \xrightarrow{a} q.$$

The derivation rules in Table 2.4.24 on Page 43 yield that there must be a P/T-net component C in S and a closed term $r \in \mathcal{C}(\mathrm{PA}^*_{\delta\lambda}+\mathrm{RN})$ such that

$$q \equiv r \parallel (\| D : D \in S \wedge D \neq C : \rho_f(\mathcal{C}^g[\![D]\!]))$$

and

$$\rho_f(\mathcal{C}^g[\![C]\!]) \xrightarrow{a} r.$$

It follows from again the rules in Table 2.4.24 that there must be an action $b \in \boldsymbol{A}$ with $\boldsymbol{f}(b) = a$ and a closed term $u \in \mathcal{C}(\mathrm{PA}^*_{\delta\lambda}+\mathrm{RN})$ such that

$$r \equiv \rho_f(u)$$

and
$\mathcal{C}^g[\![C]\!] \xrightarrow{b} u.$
As a result, Theorem 3.4.8 on Page 68 yields that there must be a P/T-net component $C' \in \mathcal{C}$ such that $u \equiv \mathcal{C}^g[\![C']\!]$
and
$C\ [b\rangle\ C'.$
Hence, a consequence of Definition 3.3.8 (Firing rule for P/T-net components) on Page 64 is that there is a transition t in C such that $C[t\rangle$ and $b = \alpha^g(t)$. Recall that $a = \boldsymbol{f}(b)$. This means that $a = \boldsymbol{f}(\alpha^g(t)) = (\bar{f}(\boldsymbol{it}), \bar{f}(\boldsymbol{ot})) = \mu^g(t)$. As in the first part of the proof, it follows that $\boldsymbol{ca} \restriction I = f(\boldsymbol{oit}) \restriction I$ and $\boldsymbol{pa} \restriction I = f(\boldsymbol{oot}) \restriction I$. Since $\boldsymbol{ca} \restriction I \leq s$ and $s' = s - \boldsymbol{ca} \restriction I \uplus \boldsymbol{pa} \restriction I$, it follows that $f(\boldsymbol{oit}) \restriction I \leq s$ and $s' = s - f(\boldsymbol{oit}) \restriction I \uplus f(\boldsymbol{oot}) \restriction I$. Since $C[t\rangle$, Definition 3.5.4 (Transition enabling in modular P/T nets) on Page 77 implies that $(M, s)[t\rangle$. As a result of Definition 3.5.5 (Firing rule) on Page 77,
$(M, s)\ [a\rangle\ (M', s'),$
where $M' = (S', P, I, f)$ with S' equal to S with C replaced by C'. Summarizing the above results means that
$p \equiv \lambda^I_{s'}(\|\ D :\ D \in S' :\ \rho_f(\mathcal{C}^g[\![D]\!])).$
Definition 3.6.1 on Page 83 (Algebraic semantics) yields that
$p \equiv \mathcal{M}^g[\![(M', s')]\!],$
which completes this part of the proof.

iii) Assume $\mathcal{M}^g[\![(M, s)]\!] \xrightarrow{a} \surd$. According to Definition 3.6.1 (Algebraic semantics), this means that
$\lambda^I_s(\|\ D :\ D \in S :\ \rho_f(\mathcal{C}^g[\![D]\!])) \xrightarrow{a} \surd.$
The derivation rules in Tables 3.4.2 on Page 67 and 2.4.24 on Page 43 imply that there is an action $b \in \boldsymbol{A}$ with $\boldsymbol{f}(b) = a$ and a component C in S such that
$\mathcal{C}^g[\![C]\!] \xrightarrow{b} \surd.$
However, this contradicts Theorem 3.4.8 on Page 68. Hence,
$\mathcal{M}^g[\![(M, s)]\!] \not\xrightarrow{a} \surd,$
which completes the proof.

□

Example 3.6.3. Let $(M, \mathbf{0})$ be the modular P/T net of Figure 3.5.2 on Page 76. This example shows that the algebraic semantics for the glass-box behavior of $(M, \mathbf{0})$ is equal to the expression derived in Example 3.4.10 on Page 70 for the glass-box behavior of the P/T-net component of Figure 3.3.1 on Page 62. This result may be expected, since the component of Figure 3.3.1 is the unfolding of $(M, \mathbf{0})$.

First, the behavior of the two components *Control* and *Processing* must be determined. It is not difficult to verify that the glass-box behavior of these components can be expressed by the following terms. The derivations are similar to the derivations in Example 3.4.10 on Page 70. As usual, transition identifiers are used to denote actions.

$\mathcal{C}^g[\![Control]\!] = (rcmd \cdot sprdy) * \delta,$ and
$\mathcal{C}^g[\![Processing]\!] = (pmat \cdot omat) * \delta.$

Second, using these expressions, the behavior of the modular net is calculated. Let I be the set of internal places $\{tcmd, tprdy\}$. Let f be the place-fusion function given in Example 3.5.3 on Page 76. Function $\boldsymbol{f}$ is the renaming function based on function f as defined before Definition 3.6.1 on Page 83. An upright transition identifier denotes the action corresponding to the transition after renaming it by means of function $\boldsymbol{f}$. For example, $\mathrm{rcmd} = \boldsymbol{f}([cicmd, rdy], [cocmd, bsy]) = ([cmd, rdy], [tcmd, bsy])$. The following equalities give the behavior of the components *Control* and *Processing* in the context of the modular net $(M, \mathbf{0})$. They are a straightforward result of the axioms for the renaming operator given in Tables 2.4.19 on Page 40 and 2.4.47 on Page 52.

$$\rho_f(\mathcal{C}^g[\![Control]\!]) = (\mathrm{rcmd} \cdot \mathrm{sprdy}) \,^*\, \delta, \quad \text{and}$$
$$\rho_f(\mathcal{C}^g[\![Processing]\!]) = (\mathrm{pmat} \cdot \mathrm{omat}) \,^*\, \delta.$$

The glass-box behavior of the modular net $(M, \mathbf{0})$ is now calculated as follows.

$$
\begin{array}{ll}
 & \mathcal{M}^g[\![(M, \mathbf{0})]\!] \\
= & \quad \{ \text{ Definition 3.6.1 (Algebraic semantics) on Page 83 } \} \\
 & \lambda_{\mathbf{0}}((\mathrm{rcmd} \cdot \mathrm{sprdy}) \,^*\, \delta \parallel (\mathrm{pmat} \cdot \mathrm{omat}) \,^*\, \delta) \\
= & \quad \{ \; M1, BKS1(2\times), A6(2\times) \; \} \\
 & \lambda_{\mathbf{0}}(\;\; \mathrm{rcmd} \cdot \mathrm{sprdy} \cdot ((\mathrm{rcmd} \cdot \mathrm{sprdy}) \,^*\, \delta) \;\lfloor\!\lfloor\; (\mathrm{pmat} \cdot \mathrm{omat}) \,^*\, \delta \\
 & \quad + \mathrm{pmat} \cdot \mathrm{omat} \cdot ((\mathrm{pmat} \cdot \mathrm{omat}) \,^*\, \delta) \;\lfloor\!\lfloor\; (\mathrm{rcmd} \cdot \mathrm{sprdy}) \,^*\, \delta \\
 & \quad) \\
= & \quad \{ \; M3(2\times) \; \} \\
 & \lambda_{\mathbf{0}}(\;\; \mathrm{rcmd} \cdot (\mathrm{sprdy} \cdot ((\mathrm{rcmd} \cdot \mathrm{sprdy}) \,^*\, \delta) \parallel (\mathrm{pmat} \cdot \mathrm{omat}) \,^*\, \delta) \\
 & \quad + \mathrm{pmat} \cdot (\mathrm{omat} \cdot ((\mathrm{pmat} \cdot \mathrm{omat}) \,^*\, \delta) \parallel (\mathrm{rcmd} \cdot \mathrm{sprdy}) \,^*\, \delta) \\
 & \quad) \\
= & \quad \{ \; CSO5, CSO4(2\times) \; \} \\
 & \quad \lambda_{\mathbf{0}}(\mathrm{rcmd}) \cdot \lambda_{[tcmd]}(\mathrm{sprdy} \cdot ((\mathrm{rcmd} \cdot \mathrm{sprdy}) \,^*\, \delta) \parallel (\mathrm{pmat} \cdot \mathrm{omat}) \,^*\, \delta) \\
 & + \lambda_{\mathbf{0}}(\mathrm{pmat}) \cdot \lambda_{\mathbf{0}}(\mathrm{omat} \cdot ((\mathrm{pmat} \cdot \mathrm{omat}) \,^*\, \delta) \parallel (\mathrm{rcmd} \cdot \mathrm{sprdy}) \,^*\, \delta) \\
= & \quad \{ \; CSO2, CSO3, A7, A6 \; \} \\
 & \mathrm{rcmd} \cdot \lambda_{[tcmd]}(\mathrm{sprdy} \cdot ((\mathrm{rcmd} \cdot \mathrm{sprdy}) \,^*\, \delta) \parallel (\mathrm{pmat} \cdot \mathrm{omat}) \,^*\, \delta) \\
= & \quad \{ \text{ Continue as above } \} \\
 & \mathrm{rcmd} \cdot \mathrm{pmat} \cdot \mathrm{omat} \cdot \mathrm{sprdy} \cdot \lambda_{\mathbf{0}}((\mathrm{rcmd} \cdot \mathrm{sprdy}) \,^*\, \delta \parallel (\mathrm{pmat} \cdot \mathrm{omat}) \,^*\, \delta) \\
= & \quad \{ \text{ Definition 3.6.1 (Algebraic semantics) on Page 83 } \} \\
 & \mathrm{rcmd} \cdot \mathrm{pmat} \cdot \mathrm{omat} \cdot \mathrm{sprdy} \cdot \mathcal{M}^g[\![(M, \mathbf{0})]\!]
\end{array}
$$

Hence, RSP^* yields

$$\mathcal{M}^g[\![(M, \mathbf{0})]\!] = (\mathrm{rcmd} \cdot \mathrm{pmat} \cdot \mathrm{omat} \cdot \mathrm{sprdy}) \,^*\, \delta.$$

This results shows that indeed the algebraic semantics for the glass-box behavior of the modular net of Figure 3.5.2 on Page 76 is equal to the glass-box behavior of the P/T-net component of Figure 3.3.1 on Page 62 as it is derived in Example 3.4.10 on Page 70.

3.6.2 System semantics

The algebraic semantics for the system behavior of modular P/T nets is defined in the process algebra $\mathrm{PA}^{\tau*}_{\delta\lambda}$+RN. The basic idea of the definition is the same as for Definition 3.6.1 on Page 83, where the algebraic glass-box semantics for modular P/T nets is defined. The only difference is that the black-box semantics for P/T-nets components is used instead of their glass-box semantics.

Definition 3.6.4. (Algebraic system semantics) Let (M, s) with $M = (S, P, I, f)$ be a modular P/T net in $\boldsymbol{\mathcal{M}}$. The algebraic semantics for the system behavior of (M, s), denoted $\mathcal{M}^s[\![(M, s)]\!]$, is defined as follows:

$\mathcal{M}^s[\![(M, s)]\!] = \lambda^I_s(\| \, C : C \in S : \rho_f(\mathcal{C}^b[\![C]\!]))$.

Theorem 3.6.5. *Assume that the semantics for modular P/T nets is the operational system semantics $\boldsymbol{\mathcal{S}}^s(\boldsymbol{\mathcal{M}})$ of Definition 3.5.8 on Page 78. For any two modular P/T nets (M, s) with $M = (S, P, I, f)$ and (M', s') in $\boldsymbol{\mathcal{M}}$, closed term p in $\mathcal{C}(\mathrm{PA}^{\tau*}_{\delta\lambda}+\mathrm{RN})$, and action $a \in \boldsymbol{A}$,*

i) $(M, s)\ [a\rangle\ (M', s') \Rightarrow \mathcal{M}^s[\![(M, s)]\!] \xrightarrow{a} \mathcal{M}^s[\![(M', s')]\!]$,

ii) $\mathcal{M}^s[\![(M, s)]\!] \xrightarrow{a} p \Rightarrow$
$(\exists\, (M', s') : (M', s') \in \boldsymbol{\mathcal{M}} : p \equiv \mathcal{M}^s[\![(M', s')]\!] \wedge (M, s)\ [a\rangle\ (M', s'))$,

iii) $\mathcal{M}^s[\![(M, s)]\!] \stackrel{a}{\not\longrightarrow} \surd$.

Proof. The proof is almost identical to the proof of Theorem 3.6.2 on Page 83. The main difference is in the use of the action functions. Furthermore, Theorem 3.4.17 on Page 73 is used instead of Theorem 3.4.8 on Page 68. □

Example 3.6.6. Let $(M, \mathbf{0})$ be the modular P/T net of Figure 3.5.2 on Page 76. An algebraic expression for the system behavior of $(M, \mathbf{0})$ is calculated in two steps. First, the black-box behavior of the two components *Control* and *Processing* is determined. Second, the results of the first step are used to calculate the system behavior in a compositional way. It is not difficult to verify that the black-box behavior of components *Control* and *Processing* is expressed by the following terms. Using the expressions for the glass-box behavior of the two components given in Example 3.6.3 on Page 85, the calculations are similar to those in Example 3.4.18 on Page 75.

$\mathcal{C}^b[\![Control]\!] = (([cmd], \mathbf{0}) \cdot (\mathbf{0}, [prdy]))^* \delta$, and
$\mathcal{C}^b[\![Processing]\!] = (([imt^3], \mathbf{0}) \cdot (\mathbf{0}, [omt]))^* \delta$.

A derivation similar to the one in Example 3.6.3 yields the following result for the system behavior of $(M, \mathbf{0})$.

$\mathcal{M}^s[\![(M, \mathbf{0})]\!] =$
$(([cmd], [tcmd]) \cdot ([imt^3, tcmd], \mathbf{0}) \cdot (\mathbf{0}, [tprdy, omt]) \cdot ([tprdy], [prdy]))^* \delta$.

3.6.3 Black-box semantics

The algebraic semantics for the black-box behavior of modular P/T nets is defined in the process algebra $\mathrm{PA}^{\tau*}_{\delta\lambda}$+RN. The black-box semantics of a modular P/T net is defined by hiding the internal consumptions and productions of the modular net in its system semantics.

Definition 3.6.7. (Algebraic black-box semantics) Let (M, s) with $M = (S, P, I, f)$ be a modular P/T net in $\boldsymbol{\mathcal{M}}$. The algebraic semantics for the black-box behavior of (M, s), denoted $\mathcal{M}^b[\![(M, s)]\!]$, is defined as follows:

$\mathcal{M}^b[\![(M, s)]\!] = \tau_I(\mathcal{M}^s[\![(M, s)]\!])$.

Theorem 3.6.8. *Assume that the semantics for modular P/T nets is the operational black-box semantics $\boldsymbol{\mathcal{S}}^b(\boldsymbol{\mathcal{M}})$ of Definition 3.5.9 on Page 79. For any two modular nets (M, s) with $M = (S, P, I, f)$ and (M', s') in $\boldsymbol{\mathcal{M}}$, closed term p in $\mathcal{C}(\mathrm{PA}^{\tau*}_{\delta\lambda} + \mathrm{RN})$, and action $a \in \boldsymbol{A}$,*

i) $(M, s)\ [a\rangle\ (M', s') \Rightarrow \mathcal{M}^b[\![(M, s)]\!] \xrightarrow{a} \mathcal{M}^b[\![(M', s')]\!]$,

ii) $\mathcal{M}^b[\![(M, s)]\!] \xrightarrow{a} p \Rightarrow$
$(\exists\, (M', s') : (M', s') \in \boldsymbol{\mathcal{M}} : p \equiv \mathcal{M}^b[\![(M', s')]\!] \wedge (M, s)\ [a\rangle\ (M', s'))$,

iii) $\mathcal{M}^b[\![(M, s)]\!] \overset{a}{\not\longrightarrow} \surd$.

Proof. As for Theorem 3.6.5 on the preceding page, the proof is almost identical to the proof of Theorem 3.6.2 on Page 83. Again the main differences are the action functions and the use of Theorem 3.4.17 on Page 73 instead of Theorem 3.4.8 on Page 68. □

Example 3.6.9. Consider again the modular net $(M, \mathbf{0})$ of Figure 3.5.2 on Page 76. Let I be the set of internal places $\{tcmd, tprdy\}$. Using the result of Example 3.6.6 on the previous page, it is not difficult to verify the following result:

$\mathcal{M}^b[\![(M, \mathbf{0})]\!] = \tau_I(\mathcal{M}^s[\![(M, \mathbf{0})]\!]) =$
$(([cmd], \mathbf{0}) \cdot ([imt^3], \mathbf{0}) \cdot (\mathbf{0}, [omt]) \cdot (\mathbf{0}, [prdy]))^* \delta$.

3.7 Case Study: The Alternating-Bit Protocol

3.7.1 Introduction

The previous four sections have introduced P/T-net components and modular P/T nets, as well as their algebraic semantics. The theory forms the basis for a formalism to design concurrent systems using both Petri nets and process algebra. Section 3.2 has briefly introduced a design method which can be used as a framework to support the integrated formalism. In this section, the method of Section 3.2 is applied to the design of a very simple communication protocol, namely the Alternating-Bit Protocol, abbreviated ABP. Since the ABP is a relatively simple and well-understood protocol, it is well suited to serve as a first case study to validate the design method and the theory introduced in the previous sections. The design and analysis of the ABP clearly shows the possibilities of the combined theory. However, the ABP is not a truly concurrent system. Its behavior is strictly sequential. In the next section, a case study concerning a real concurrent system is presented. The system designed in that case study is more complex than the ABP and its verification reveals some of the limitations of the theory of this chapter.

Informal system requirements. The goal is to design a protocol which guarantees the correct transmission of messages over a network with unreliable channels. Channels are point-to-point, unidirectional, and message delivery is FIFO. A channel may corrupt messages, but it cannot lose or duplicate them. The receiver is able to detect whether a message is corrupted or not. It is assumed that any two sites in the network are connected by channels in both directions. It may also be assumed that a channel once in a while transmits a message without corrupting it.

A language for behavioral specifications. As explained in Section 3.2, it is often useful to give a specification of the communication behavior of the system under development in terms of a process algebra or a simple language based on the principles of process algebra. Since it must be possible to verify the algebraic characterization of the behavior of a system's net model against such a specification, any of the process algebras defined in Sections 3.4 and 3.6 could be used as a specification language. However, these process algebras have the disadvantage that they contain several operators that are generally not suitable for specifications. Examples of such operators are the left merge and the causal state operator. In addition, the format of the multi-actions in these process algebras is not very suitable for the purpose of specifications. Often, actions of a system can be modeled as the consumption or production of a single token. It is tedious if one has to write $([t], \mathbf{0})$ simply to say that a single token t is consumed. Therefore, a simple language for behavioral specifications is introduced, which is sufficient for the case studies in the remainder of this chapter.

A *specification* can be one of the following: an inaction, an action, a choice, a sequential composition, a parallel composition, or an iteration.

$$
\begin{array}{lcl}
\mathit{specification} & ::= & \text{“}\delta\text{”} \\
& | & \mathit{action} \\
& | & \mathit{specification} \text{ “ } + \text{ ” } \mathit{specification} \\
& | & \mathit{specification} \text{ “ } \cdot \text{ ” } \mathit{specification} \\
& | & \mathit{specification} \text{ “ } \| \text{ ” } \mathit{specification} \\
& | & \mathit{specification} \text{ “}{*}\text{” } \mathit{specification}
\end{array}
$$

The binding precedence of the operators used to construct *specifications* is the same as the binding precedence of the corresponding process-algebraic operators.

An *action* is a silent action, the consumption of a token, the production of a token, or a multi-action consisting of the simultaneous consumption and/or production of multiple tokens. The silent action is included because sometimes it can be convenient to specify that a system may perform some internal actions. A consumption is denoted by postfixing a token identifier with a question mark; a production is denoted by means of an exclamation mark. Recall that U is the universe of identifiers from which also token identifiers are taken.

$$
\begin{array}{lcl}
\mathit{action} & ::= & \text{“}\tau\text{”} \\
& | & U\text{“?”} \\
& | & U\text{“!”} \\
& | & \mathit{action} \text{ “ } | \text{ ” } \mathit{action}
\end{array}
$$

The unary operators ? and ! on *actions* bind stronger than the multi-action operator |. The binding precedence of the multi-action operator is higher than the precedence of any of the operators on *specifications*.

For the purpose of verifications, an interpretation of *specifications* as process-algebraic terms is needed.

Definition 3.7.1. (Algebraic interpretation of specifications) The interpretation $[\![\_]\!]$ of *specifications* as closed PA^*_δ terms is defined as follows. Let t be a token identifier in U; let a_0 and a_1 be *actions* and s_0 and s_1 *specifications*.

$$\begin{array}{lcl} [\![\tau]\!] & = & (\mathbf{0}, \mathbf{0}), \\ [\![t?]\!] & = & ([t], \mathbf{0}), \\ [\![t!]\!] & = & (\mathbf{0}, [t]), \\ [\![a_0 \mid a_1]\!] & = & (\boldsymbol{c}[\![a_0]\!] \uplus \boldsymbol{c}[\![a_1]\!], \boldsymbol{p}[\![a_0]\!] \uplus \boldsymbol{p}[\![a_1]\!]), \\ [\![\delta]\!] & = & \delta, \text{ and} \\ [\![s_0 \oplus s_1]\!] & = & [\![s_0]\!] \oplus [\![s_1]\!], \text{ where } \oplus \in \{+, \cdot, \|, ^*\}. \end{array}$$

3.7.2 A modular P/T net of the ABP

Behavioral specification. Consider a situation where a user, which may be anything ranging from a person to a piece of equipment or a software application, wants to transmit some messages to another user. Communication protocols that must communicate over unreliable networks are generally based on acknowledgments to guarantee that messages are transmitted properly. A very simple solution satisfying the informal system requirements above is as follows. The user wanting to transmit messages has a sender component available; the user at the receiving site has a receiver component. The protocol starts when the sender gets an input message from the user. It transmits the message over one of the channels in the network and then waits for an acknowledgment from the receiver, indicating that the message has been received and forwarded to the user at the receiving site. When such an acknowledgment is received by the sender, it is ready to receive another message from the user.

In the design process, the users are considered to be part of the environment of the protocol. From the point of view of the environment, the high-level, external behavior of the simple protocol sketched above can be specified as follows. Let i and o be identifiers in U.

$$ABP_{\mathrm{spec}} = (i? \cdot o!)^* \delta. \tag{3.7.2}$$

This specification specifies the *black-box* behavior of the protocol, meaning that it omits all the details of the actual transmission of messages. It states that the protocol starts with reading a message i from the environment. The specification abstracts away from aspects as message contents and the addresses of the sending and receiving sites. After reading a message, the protocol produces a message o, modeling the output of the receiver to the user at the receiving site. Clearly, the message contents of o should coincide with the contents of message i and it should be received by the proper site. However, since such aspects are not relevant for the correct communication behavior of the protocol, they are omitted from the specification. The iteration in the specification implies that the protocol repeats the aforementioned two steps indefinitely. This means that the specification uses the assumption that network channels cannot always corrupt a message. Eventually, every message that is sent arrives at the receiver after which the protocol is ready to send the next message.

Petri-net model. The next design step is to construct a Petri-net model of the protocol. Based on the informal system requirements, four components are distinguished, namely a sender, a receiver, and two channels connecting the sender and the receiver. Since channels are unidirectional, two channels are needed to allow communication in two directions. Figure 3.7.3 on the following page shows a modular-P/T-net model for the protocol. The sender is modeled by P/T-net component *SN* and the receiver by component *RC*. The channel which is used for sending messages from sender to receiver is modeled by component *MC*; the channel for acknowledgments in the other direction is modeled by component *AC*. The basic idea of the protocol is that a bit is used to mark messages. The sender attaches bit 0 to the first message it sends. Then, it waits until it receives an acknowledgment which also carries bit 0. If it receives a corrupted acknowledgment or an acknowledgment with bit 1, it retransmits the message with bit 0. If it receives the correct acknowledgment, it proceeds with the next message to which it attaches a 1 bit. Since the modular-P/T-net formalism is uncolored, it is necessary to use separate places for each message type. The meaning of the internal places in the model of Figure 3.7.3 on the next page is as follows.

$j \in \{0, 1\}$	
jms	: places to store messages with bit j that are ready to be sent.
jmr	: messages with bit j, after transmission and ready for receipt.
$\perp m$	: messages that are corrupted by the channel.
$jas, jar, \perp a$	: places for acknowledgments.

As mentioned in Section 3.2, in many practical applications, the net model of a system is defined using colored Petri nets. If colored nets are used, it is possible to collapse all the places between any two components in the net model of Figure 3.7.3 into a single place. Data types can be used to distinguish messages with different bits and corrupted messages. In addition, it is possible to take into account message contents. If some timing mechanism is available, it is also possible to model message delays and timeouts. It may also be possible to quantify the possibility that a channel corrupts a message.

Behavioral specification of the P/T-net components. The net model of Figure 3.7.3 on the following page is not useful without defining the behavior of the four P/T-net components in a precise way. The easiest components are the two channels. According to their informal specification on Page 89, channels either simply forward a message or they deliver a corrupt message. Since both channels behave the same, it suffices to design a single, generic component. The two channels in Figure 3.7.3 are both instances of this

Specification:
$ABP_{\text{spec}} = (i? \cdot o!)^* \delta$

ABP :

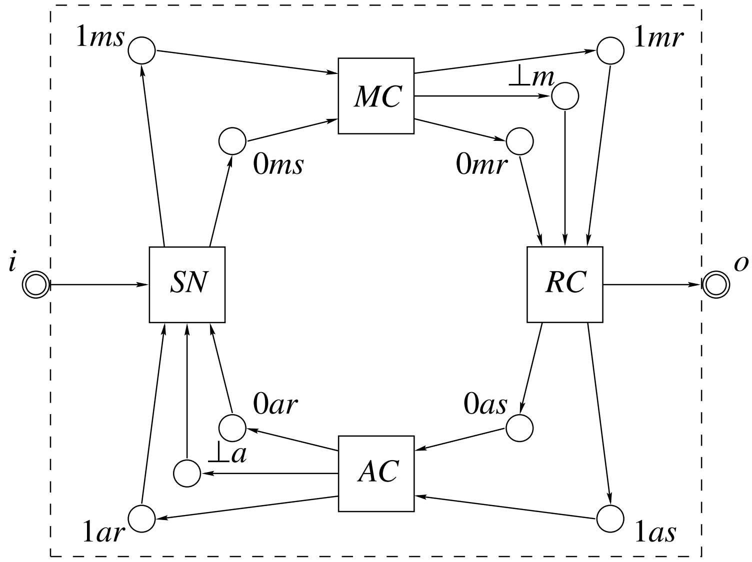


Figure 3.7.3: A modular P/T net modeling the Alternating-Bit Protocol.

one component. Note that, formally, the definition of modular P/T nets (Definition 3.5.1 on Page 75) does not allow such a construction. It requires that all places and transitions of components of a modular net are different. However, in the design of real-world applications, it is often useful – or even a *goal* – to reuse components. Using a renaming of places and transitions, it is straightforward to satisfy the requirements of Definition 3.5.1.

The interface of the P/T-net component *CH* modeling a channel contains the following pins. The two mappings f_m and f_a define the mapping of pins of components to places of the modular P/T net *ABP* for message channel *MC* and acknowledgment channel *AC*, respectively.

$j \in \{0, 1\}$		f_m	f_a
ij	: input messages with bit j	jms	jas
oj	: output messages with bit j	jmr	jar
$\perp$	: corrupted messages	$\perp m$	$\perp a$

The behavior of a channel is now specified as follows.

$$CH_{\text{spec}} = (i0? \cdot (o0! + \perp!) + i1? \cdot (o1! + \perp!))^* \delta. \tag{3.7.4}$$

The other two components are more complicated. As explained earlier, the sender *SN* attaches bit 0 to the first message it sends. Then, it waits until it receives an acknowledgment which also carries bit 0. Upon receipt of such an acknowledgment, it proceeds with the next message to which it attaches a 1 bit. However, before the correct acknowledgment is received, it may happen that the sender receives a corrupted acknowledgment or an acknowledgment with a 1 bit. In the first case, the sender does not know whether its last message has been received properly. In the second case, it knows that its message has been corrupted. Therefore, upon receipt of an incorrect acknowledgment, the sender retransmits its message and starts waiting for another acknowledgment. The sending of a 1 message goes in a very similar way. The sender repeats the transmission of a 0 message followed by a 1 message indefinitely. The interface of *SN* consists of the following pins. The mapping f_s maps the interface of *SN* to places of *ABP*.

$j \in \{0, 1\}$		f_s
di	: data input	i
jmo	: output messages with bit j	jms
$jai, \bot ai$	: input acknowledgments	$jar, \bot a$

The behavior of the sender is now specified as follows.

$$\begin{aligned} SN_{\text{spec}} = \; & (di? \mid 0mo! \cdot ((\bot ai? \mid 0mo! + 1ai? \mid 0mo!)^* \, 0ai?) \cdot \\ & \quad di? \mid 1mo! \cdot ((\bot ai? \mid 1mo! + 0ai? \mid 1mo!)^* \, 1ai?) \\ &)^* \, \delta. \end{aligned} \tag{3.7.5}$$

In this specification, the transmission or retransmission of a message is a single action. Another possibility is to specify that the (re-)transmission of a message consists of two separate actions, namely an input action and an output action, similar to the way a channel behaves. In fact, one could argue that the latter solution is more natural than the one chosen above. However, considering the fact that Petri nets are used to model the system under development, this is not necessarily true. In theory, the step in which a behavioral specification is given and the step in which a Petri-net model is constructed are nicely separated. In practice, this separation is almost never very clear. A designer usually has a Petri-net model in mind when writing the behavioral specification. Often, the specification and the net model are even developed simultaneously. Simultaneous consumptions and productions in a specification are very natural when thinking in terms of Petri nets. They simply represent the firing of a single transition in a Petri-net model. In general, specifications with as many simultaneous consumptions and productions as possible yield the simplest net models and calculations.

Similar to the sender, the receiver *RC* performs two small terminating iterations inside its main non-terminating iteration. In the beginning, the receiver is expecting a message with bit 0. As long as it receives a corrupted message or a message with bit 1, it sends an acknowledgment with bit 1 attached, which is interpreted by the sender as a negative acknowledgment. If it receives the message with bit 0, it sends an acknowledgment with bit 0, which denotes a positive acknowledgment. After the receiver has received and acknowledged a 0 message, it starts waiting for a 1 message. When it has received both a 0 and a 1 message, it starts the next cycle of its main iteration. The interface of the receiver *RC* consists of the following pins. The mapping f_r maps pins of *RC* to places of the modular net *ABP*.

$j \in \{0,1\}$		f_r
do	: data output	o
jao	: output acknowledgments	jas
jmi, $\perp mi$	: input messages	jmr, $\perp m$

The behavior of *RC* is specified as follows.

$$
\begin{aligned}
RC_{\text{spec}} = (&((\perp mi? \mid 1ao! + 1mi? \mid 1ao!)^{*}0mi? \mid 0ao! \mid do!) \cdot \\
&((\perp mi? \mid 0ao! + 0mi? \mid 0ao!)^{*}1mi? \mid 1ao! \mid do!) \\
)&^{*}\delta.
\end{aligned} \tag{3.7.6}
$$

Simulation. Given the specifications of the four components of the modular P/T net *ABP* of Figure 3.7.3 on Page 92, it is, in principle, possible to simulate *ABP* without detailing the net models of the components. However, current tools as ExSpect and Design/CPN do not allow simulation of such incomplete net models. For complex designs, simulation at different levels of abstraction could be a useful way to correct some of the more obvious mistakes in both the behavioral specifications and the modular-net model.

3.7.3 Verifying the modular P/T net

The next step in the design of a net model of the ABP is to verify the behavior of the modular net *ABP* against its specification ABP_{spec} given in Definition 3.7.2 on Page 90. It may be assumed that the four components of *ABP* are implemented according to their specifications. Since specification ABP_{spec} prescribes the black-box behavior of the modular net *ABP*, the goal is to show that the algebraic black-box semantics of *ABP* is equal to the algebraic interpretation of ABP_{spec}.

Recall that $\boldsymbol{A}$ is the set of multi-actions of Definition 3.3.7 on Page 63. On Page 83, it is explained how a place-fusion function is used to define a renaming function on actions in $\boldsymbol{A}$. Note that the combination of the four mappings f_s, f_r, f_m, and f_a given in the previous subsection forms the place-fusion function of the modular P/T net *ABP* of Figure 3.7.3 on Page 92. Let $\boldsymbol{s}$, $\boldsymbol{r}$, $\boldsymbol{m}$, and $\boldsymbol{a}$ be four renaming functions based on the four mappings f_s, f_r, f_m, and f_a. Let I be the set of internal places $\{0ms, 1ms, 0mr, 1mr, \perp m, 0as, 1as, 0ar, 1ar, \perp a\}$. According to Definitions 3.6.4 (Algebraic system semantics) on Page 87 and 3.6.7 (Algebraic black-box semantics) on Page 88, the algebraic black-box semantics of *ABP* is defined as follows.

$$\mathcal{M}^b[\![ABP]\!] = \tau_I(\mathcal{M}^s[\![ABP]\!]), \tag{3.7.7}$$

where

$$\mathcal{M}^s[\![ABP]\!] = \lambda^I_{\boldsymbol{0}}(\rho_{\boldsymbol{s}}(\mathcal{C}^b[\![SN]\!]) \parallel \rho_{\boldsymbol{m}}(\mathcal{C}^b[\![MC]\!]) \parallel \rho_{\boldsymbol{r}}(\mathcal{C}^b[\![RC]\!]) \parallel \rho_{\boldsymbol{a}}(\mathcal{C}^b[\![AC]\!])). \tag{3.7.8}$$

The four P/T-net components *SN*, *RC*, *MC*, and *AC* have not yet been defined. Hence, their algebraic semantics is not yet known. However, as mentioned, it may be assumed that the four components behave as specified. This means that, in the verification, the following equalities may be used.

$$\begin{aligned} \mathcal{C}^b[\![SN]\!] &= [\![SN_{\text{spec}}]\!], \\ \mathcal{C}^b[\![RC]\!] &= [\![RC_{\text{spec}}]\!], \text{ and} \\ \mathcal{C}^b[\![MC]\!] &= \mathcal{C}^b[\![AC]\!] = [\![CH_{\text{spec}}]\!], \end{aligned} \tag{3.7.9}$$

where SN_{spec}, RC_{spec}, and CH_{spec} are given in the previous subsection in Definitions 3.7.5, 3.7.6, and 3.7.4, respectively. To prove that the modular net *ABP* satisfies its specification, it must now be shown that

$$\mathcal{M}^b[\![ABP]\!] = [\![ABP_{\text{spec}}]\!], \tag{3.7.10}$$

where ABP_{spec} is given in Definition 3.7.2 on Page 90. Note that it follows from Definition 3.7.1 (Interpretation $[\![\_]\!]$) also on Page 90 that

$$[\![ABP_{\text{spec}}]\!] = (([i], \mathbf{0}) \cdot (\mathbf{0}, [o]))^* \delta. \tag{3.7.11}$$

The verification consists of three parts. First, the behavior of the components in the context of the modular net *ABP* is determined. Second, the results are used to calculate the system semantics of *ABP*. Third, the internal consumptions and productions of *ABP* are hidden, yielding an algebraic expression for the black-box behavior of *ABP*, which should be equal to the algebraic expression $[\![ABP_{\text{spec}}]\!]$ (Results 3.7.10 and 3.7.11).

Verification, Part 1. The first part of the verification starts with the channels. The following derivation gives an expression for the behavior of the message channel *MC* as it is embedded in the modular net *ABP* of Figure 3.7.3 on Page 92.

$$\begin{array}{ll}
 & \rho_{\boldsymbol{m}}(\mathcal{C}^b[\![MC]\!]) \\
= & \quad \{ \text{ Result 3.7.9 } (MC \text{ satisfies its specification}) \} \\
 & \rho_{\boldsymbol{m}}([\![CH_{\text{spec}}]\!]) \\
= & \quad \{ \text{ Definition 3.7.4 (Specification } CH_{\text{spec}}) \text{ on Page 92 } \} \\
 & \rho_{\boldsymbol{m}}([\![(i0? \cdot (o0! + \bot!) + i1? \cdot (o1! + \bot!))^* \delta]\!]) \\
= & \quad \{ \text{ Definition 3.7.1 (Interpretation } [\![_]\!]) \text{ on Page 90 } \} \\
 & \rho_{\boldsymbol{m}}(((([i0], \mathbf{0}) \cdot ((\mathbf{0}, [o0]) + (\mathbf{0}, [\bot])) + ([i1], \mathbf{0}) \cdot ((\mathbf{0}, [o1]) + (\mathbf{0}, [\bot])))^* \delta) \\
= & \quad \{ \ BKS4, RN1, 2, 3 \text{ (repeatedly) } \} \\
 & (\ \boldsymbol{m}([i0], \mathbf{0}) \cdot (\boldsymbol{m}(\mathbf{0}, [o0]) + \boldsymbol{m}(\mathbf{0}, [\bot])) \\
 & + \boldsymbol{m}([i1], \mathbf{0}) \cdot (\boldsymbol{m}(\mathbf{0}, [o1]) + \boldsymbol{m}(\mathbf{0}, [\bot])) \\
 &)^* \boldsymbol{m}(\delta) \\
= & \quad \{ \text{ Definition of } \boldsymbol{m} \text{ (see Page 94) } \} \\
 & (([0ms], \mathbf{0}) \cdot ((\mathbf{0}, [0mr]) + (\mathbf{0}, [\bot m])) + ([1ms], \mathbf{0}) \cdot ((\mathbf{0}, [1mr]) + (\mathbf{0}, [\bot m])))^* \delta
\end{array}$$

Summarizing,

$$\begin{aligned} &\rho_{\boldsymbol{m}}(\mathcal{C}^b[\![MC]\!]) = \\ &\quad (([0ms], \mathbf{0}) \cdot ((\mathbf{0}, [0mr]) + (\mathbf{0}, [\bot m])) + ([1ms], \mathbf{0}) \cdot ((\mathbf{0}, [1mr]) + (\mathbf{0}, [\bot m])))^* \delta. \end{aligned} \tag{3.7.12}$$

In a very similar way, it is possible to obtain an expression for the acknowledgment channel AC.

$$\rho_a(\mathcal{C}^b[\![AC]\!]) = \qquad (3.7.13)$$
$$(([0as], \mathbf{0}) \cdot ((\mathbf{0}, [0ar]) + (\mathbf{0}, [\bot a])) + ([1as], \mathbf{0}) \cdot ((\mathbf{0}, [1ar]) + (\mathbf{0}, [\bot a])))^* \delta.$$

The calculations for the sender SN and receiver RC are slightly more complicated. However, they follow the same lines and are, therefore, omitted.

$$\rho_s(\mathcal{C}^b[\![SN]\!]) = \qquad (3.7.14)$$
$$\begin{array}{l} (\ ([i], [0ms]) \cdot (\ (([\bot a], [0ms]) + ([1ar], [0ms]))^* ([0ar], \mathbf{0})\ ) \cdot \\ \quad ([i], [1ms]) \cdot (\ (([\bot a], [1ms]) + ([0ar], [1ms]))^* ([1ar], \mathbf{0})\ ) \\ )^* \delta \text{ and} \end{array}$$

$$\begin{array}{l} \rho_r(\mathcal{C}^b[\![RC]\!]) = ((\ (([\bot m], [1as]) + ([1mr], [1as]))^* ([0mr], [0as, o])\ ) \cdot \\ \qquad\qquad (\ (([\bot m], [0as]) + ([0mr], [0as]))^* ([1mr], [1as, o])\ ) \\ \qquad)^* \delta. \end{array} \qquad (3.7.15)$$

Verification, Part 2. The second part of the verification is the most complicated part. Recall that the goal of the second part is to calculate an expression for the system behavior of the modular net ABP. To improve the readability, actions are abbreviated as shown in Table 3.7.16 on the next page. In addition, Table 3.7.17 shows some abbreviations of behaviors. The complement function $\bar{\ }$ on bits is defined as expected: $\bar{0}$ equals 1 and $\bar{1}$ equals 0.

Recall Result 3.7.8 on Page 94 for the system behavior of ABP. Using the abbreviations of Tables 3.7.16 and 3.7.17 and Results 3.7.12 through 3.7.15 of the first part of the verification, the following equation is obtained.

$$\mathcal{M}^s[\![ABP]\!] = \lambda^I_{\mathbf{0}}(SMRA). \qquad (3.7.18)$$

In the derivation below, the superscript I of the state operator is omitted. The derivation shows how to compute the first action that can be performed by ABP.

$$\begin{array}{ll} & \lambda^I_{\mathbf{0}}(SMRA) \\ = & \quad \{ \text{ Expand abbreviations } \} \\ & \lambda_{\mathbf{0}}(S^*\delta \parallel M^*\delta \parallel R^*\delta \parallel A^*\delta) \\ = & \quad \{ \text{ Expansion (see Page 47) } \} \\ & \lambda_{\mathbf{0}}(S^*\delta \mathbin{⌊\!\!⌊} MRA + M^*\delta \mathbin{⌊\!\!⌊} SRA + R^*\delta \mathbin{⌊\!\!⌊} SMA + A^*\delta \mathbin{⌊\!\!⌊} SMR) \\ = & \quad \{ BKS1, A6 \text{ (repeatedly) } \} \\ & \lambda_{\mathbf{0}}(S \cdot (S^*\delta) \mathbin{⌊\!\!⌊} MRA + M \cdot (M^*\delta) \mathbin{⌊\!\!⌊} SRA + R \cdot (R^*\delta) \mathbin{⌊\!\!⌊} SMA + A \cdot (A^*\delta) \mathbin{⌊\!\!⌊} SMR) \\ = & \quad \{ \text{ Expand abbreviations } \} \\ & \lambda_{\mathbf{0}}\ (\ \mathrm{0t} \cdot ((\mathrm{0rt_1} + \mathrm{0rt_2})^* \mathrm{0a}) \cdot S1 \cdot (S^*\delta) \mathbin{⌊\!\!⌊} MRA \\ & \quad + (\mathrm{gm0} \cdot (\mathrm{fm0} + \mathrm{cm}) + \mathrm{gm1} \cdot (\mathrm{fm1} + \mathrm{cm})) \cdot (M^*\delta) \mathbin{⌊\!\!⌊} SRA \\ & \quad + ((\mathrm{1na_1} + \mathrm{1na_2})^* \mathrm{0pa}) \cdot R1 \cdot (R^*\delta) \mathbin{⌊\!\!⌊} SMA \\ & \quad + (\mathrm{ga0} \cdot (\mathrm{fa0} + \mathrm{ca}) + \mathrm{ga1} \cdot (\mathrm{fa1} + \mathrm{ca})) \cdot (A^*\delta) \mathbin{⌊\!\!⌊} SMR \\ & \quad) \end{array}$$

$j \in \{0, 1\}$		
Message Channel		
$([jms], \mathbf{0})$	gm$j$	Get a j message from the sender.
$(\mathbf{0}, [jmr])$	fm$j$	Forward a j message to the receiver.
$(\mathbf{0}, [\bot m])$	cm	Corrupt a message.
Acknowledgment Channel		
$([jas], \mathbf{0})$	ga$j$	Get a j acknowledgment from the receiver.
$(\mathbf{0}, [jar])$	fa$j$	Forward a j acknowledgment to the sender.
$(\mathbf{0}, [\bot a])$	ca	Corrupt an acknowledgment.
Sender		
$([i], [jms])$	jt	Read input and transmit a j message.
$([\bot a], [jms])$	$j\mathrm{rt}_1$	Retransmit a j message.
$([\bar{j}ar], [jms])$	$j\mathrm{rt}_2$	Retransmit a j message.
$([jar], \mathbf{0})$	ja	Receive an acknowledgment for a j message.
Receiver		
$([\bot m], [jas])$	$j\mathrm{na}_1$	Send a negative j acknowledgment.
$([jmr], [jas])$	$j\mathrm{na}_2$	Send a negative j acknowledgment.
$([jmr], [jas, o])$	$j$pa	Send a positive j acknowledgment and output the received message.

Table 3.7.16: Abbreviations of actions of the modular net *ABP*.

$j \in \{0, 1\}$		
Channels		
$\mathrm{gm}j \cdot (\mathrm{fm}j + \mathrm{cm})$	Mj	Forward a j message from sender to receiver.
$M0 + M1$	M	Forward a message.
$\mathrm{ga}j \cdot (\mathrm{fa}j + \mathrm{ca})$	Aj	Forward a j acknowledgment.
$A0 + A1$	A	Forward an acknowledgment.
Sender/Receiver		
$j\mathrm{t} \cdot ((j\mathrm{rt}_1 + j\mathrm{rt}_2) * j\mathrm{a})$	Sj	Send a j message.
$S0 \cdot S1$	S	One main sender cycle.
$(\bar{j}\mathrm{na}_1 + \bar{j}\mathrm{na}_2) * j\mathrm{pa}$	Rj	Receive a j message.
$R0 \cdot R1$	R	One main receiver cycle.
All components		
$W, X, Y, Z \in \{M, A, S, R\}$		
$W * \delta \parallel X * \delta \parallel Y * \delta \parallel Z * \delta$	$WXYZ$	
$X * \delta \parallel Y * \delta \parallel Z * \delta$	XYZ	
$X * \delta \parallel Y * \delta$	XY	

Table 3.7.17: Abbreviations of partial behaviors of the modular net *ABP*.

$$
\begin{aligned}
= \quad & \{\ BKS1\ \} \\
& \lambda_{\mathbf{0}}\ (\ 0\mathrm{t} \cdot ((0\mathrm{rt}_1 + 0\mathrm{rt}_2)\,{}^*\,0\mathrm{a}) \cdot S1 \cdot (S\,{}^*\,\delta) \mathbin{\|\!\_} MRA \\
& \quad + (\mathrm{gm}0 \cdot (\mathrm{fm}0 + \mathrm{cm}) + \mathrm{gm}1 \cdot (\mathrm{fm}1 + \mathrm{cm})) \cdot (M\,{}^*\,\delta) \mathbin{\|\!\_} SRA \\
& \quad + ((1\mathrm{na}_1 + 1\mathrm{na}_2) \cdot ((1\mathrm{na}_1 + 1\mathrm{na}_2)\,{}^*\,0\mathrm{pa}) + 0\mathrm{pa}) \cdot R1 \cdot (R\,{}^*\,\delta) \mathbin{\|\!\_} SMA \\
& \quad + (\mathrm{ga}0 \cdot (\mathrm{fa}0 + \mathrm{ca}) + \mathrm{ga}1 \cdot (\mathrm{fa}1 + \mathrm{ca})) \cdot (A\,{}^*\,\delta) \mathbin{\|\!\_} SMR \\
& \quad) \\
= \quad & \{\ A4,\ M4 \text{ (repeatedly)}\ \} \\
& \lambda_{\mathbf{0}}\ (\ 0\mathrm{t} \cdot ((0\mathrm{rt}_1 + 0\mathrm{rt}_2)\,{}^*\,0\mathrm{a}) \cdot S1 \cdot (S\,{}^*\,\delta) \mathbin{\|\!\_} MRA \\
& \quad + \mathrm{gm}0 \cdot (\mathrm{fm}0 + \mathrm{cm}) \cdot (M\,{}^*\,\delta) \mathbin{\|\!\_} SRA \\
& \quad + \mathrm{gm}1 \cdot (\mathrm{fm}1 + \mathrm{cm}) \cdot (M\,{}^*\,\delta) \mathbin{\|\!\_} SRA \\
& \quad + 1\mathrm{na}_1 \cdot ((1\mathrm{na}_1 + 1\mathrm{na}_2)\,{}^*\,0\mathrm{pa}) \cdot R1 \cdot (R\,{}^*\,\delta) \mathbin{\|\!\_} SMA \\
& \quad + 1\mathrm{na}_2 \cdot ((1\mathrm{na}_1 + 1\mathrm{na}_2)\,{}^*\,0\mathrm{pa}) \cdot R1 \cdot (R\,{}^*\,\delta) \mathbin{\|\!\_} SMA \\
& \quad + 0\mathrm{pa} \cdot R1 \cdot (R\,{}^*\,\delta) \mathbin{\|\!\_} SMA \\
& \quad + \mathrm{ga}0 \cdot (\mathrm{fa}0 + \mathrm{ca}) \cdot (A\,{}^*\,\delta) \mathbin{\|\!\_} SMR \\
& \quad + \mathrm{ga}1 \cdot (\mathrm{fa}1 + \mathrm{ca}) \cdot (A\,{}^*\,\delta) \mathbin{\|\!\_} SMR \\
& \quad) \\
= \quad & \{\ M3 \text{ (repeatedly)}\ \} \\
& \lambda_{\mathbf{0}}\ (\ 0\mathrm{t} \cdot (((0\mathrm{rt}_1 + 0\mathrm{rt}_2)\,{}^*\,0\mathrm{a}) \cdot S1 \cdot (S\,{}^*\,\delta) \parallel MRA) \\
& \quad + \mathrm{gm}0 \cdot ((\mathrm{fm}0 + \mathrm{cm}) \cdot (M\,{}^*\,\delta) \parallel SRA) \\
& \quad + \mathrm{gm}1 \cdot ((\mathrm{fm}1 + \mathrm{cm}) \cdot (M\,{}^*\,\delta) \parallel SRA) \\
& \quad + 1\mathrm{na}_1 \cdot (((1\mathrm{na}_1 + 1\mathrm{na}_2)\,{}^*\,0\mathrm{pa}) \cdot R1 \cdot (R\,{}^*\,\delta) \parallel SMA) \\
& \quad + 1\mathrm{na}_2 \cdot (((1\mathrm{na}_1 + 1\mathrm{na}_2)\,{}^*\,0\mathrm{pa}) \cdot R1 \cdot (R\,{}^*\,\delta) \parallel SMA) \\
& \quad + 0\mathrm{pa} \cdot (R1 \cdot (R\,{}^*\,\delta) \parallel SMA) \\
& \quad + \mathrm{ga}0 \cdot ((\mathrm{fa}0 + \mathrm{ca}) \cdot (A\,{}^*\,\delta) \parallel SMR) \\
& \quad + \mathrm{ga}1 \cdot ((\mathrm{fa}1 + \mathrm{ca}) \cdot (A\,{}^*\,\delta) \parallel SMR) \\
& \quad) \\
= \quad & \{\ CSO5,\ 4 \text{ (repeatedly)}\ \} \\
& \quad \lambda_{\mathbf{0}}(0\mathrm{t}) \cdot \lambda_{[0ms]}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2)\,{}^*\,0\mathrm{a}) \cdot S1 \cdot (S\,{}^*\,\delta) \parallel MRA) \\
& + \lambda_{\mathbf{0}}(\mathrm{gm}0) \cdot \lambda_{\mathbf{0}}((\mathrm{fm}0 + \mathrm{cm}) \cdot (M\,{}^*\,\delta) \parallel SRA) \\
& + \lambda_{\mathbf{0}}(\mathrm{gm}1) \cdot \lambda_{\mathbf{0}}((\mathrm{fm}1 + \mathrm{cm}) \cdot (M\,{}^*\,\delta) \parallel SRA) \\
& + \lambda_{\mathbf{0}}(1\mathrm{na}_1) \cdot \lambda_{[1as]}(((1\mathrm{na}_1 + 1\mathrm{na}_2)\,{}^*\,0\mathrm{pa}) \cdot R1 \cdot (R\,{}^*\,\delta) \parallel SMA) \\
& + \lambda_{\mathbf{0}}(1\mathrm{na}_2) \cdot \lambda_{[1as]}(((1\mathrm{na}_1 + 1\mathrm{na}_2)\,{}^*\,0\mathrm{pa}) \cdot R1 \cdot (R\,{}^*\,\delta) \parallel SMA) \\
& + \lambda_{\mathbf{0}}(0\mathrm{pa}) \cdot \lambda_{[0as]}(R1 \cdot (R\,{}^*\,\delta) \parallel SMA) \\
& + \lambda_{\mathbf{0}}(\mathrm{ga}0) \cdot \lambda_{\mathbf{0}}((\mathrm{fa}0 + \mathrm{ca}) \cdot (A\,{}^*\,\delta) \parallel SMR) \\
& + \lambda_{\mathbf{0}}(\mathrm{ga}1) \cdot \lambda_{\mathbf{0}}((\mathrm{fa}1 + \mathrm{ca}) \cdot (A\,{}^*\,\delta) \parallel SMR) \\
= \quad & \{\ CSO2,\ 3,\ A7 \text{ (repeatedly)}\ \} \\
& 0\mathrm{t} \cdot \lambda_{[0ms]}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2)\,{}^*\,0\mathrm{a}) \cdot S1 \cdot (S\,{}^*\,\delta) \parallel MRA) + \delta + \delta + \delta + \delta + \delta + \delta + \delta \\
= \quad & \{\ A6 \text{ (repeatedly)}\ \} \\
& 0\mathrm{t} \cdot \lambda_{[0ms]}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2)\,{}^*\,0\mathrm{a}) \cdot S1 \cdot (S\,{}^*\,\delta) \parallel MRA)
\end{aligned}
$$

Hence, the first action of *ABP* is, not surprisingly, the transmission of a message with bit 0 attached (action 0t). The remaining behavior of *ABP* is given by the algebraic term $\lambda_{[0ms]}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2)\,{}^*\,0\mathrm{a}) \cdot S1 \cdot (S\,{}^*\,\delta) \parallel MRA)$. By means of very similar calculations, it is possible to compute all the following actions. Essentially, one just repeatedly applies the expansion theorem and the axioms of the causal state operator to determine which actions can be executed and which actions cannot. Eventually, *ABP* returns to its initial state, which means that the complete behavior is calculated. The calculations are tedious, but not very complex. Hence, they are omitted. In the following, the results of the verification are summarized.

One cycle in the main loop of *ABP* consists of the successful transmission of two messages, one with bit 0 attached and one with bit 1 attached. If no messages or acknowledgments are corrupted, such a cycle consists of fourteen actions. Consequently, the protocol passes through fourteen different states. Each state is characterized by an algebraic term. The initial state is characterized by the term $\mathcal{M}^s[\![ABP]\!]$. The thirteen other states are denoted by the abbreviations X_1 through X_{13}. The meaning of these abbreviations is given in Definition 3.7.20. To clarify the results given below, the operational semantics for the algebraic system behavior of *ABP*, $\mathcal{M}^s[\![ABP]\!]$, is depicted in Figure 3.7.21 on Page 101. The main cycle of *ABP* is summarized in Result 3.7.19. At four points, it is possible that messages or acknowledgments are corrupted. The resulting states X_{21}, X_{51}, X_{91}, and X_{121} are treated below.

$$\begin{aligned}
\mathcal{M}^s[\![ABP]\!] &= \text{0t}\cdot X_1,\\
X_1 &= \text{gm0}\cdot X_2,\\
X_2 &= \text{fm0}\cdot X_3+\text{cm}\cdot X_{21},\\
X_3 &= \text{0pa}\cdot X_4,\\
X_4 &= \text{ga0}\cdot X_5,\\
X_5 &= \text{fa0}\cdot X_6+\text{ca}\cdot X_{51},\\
X_6 &= \text{0a}\cdot X_7,\\
X_7 &= \text{1t}\cdot X_8,\\
X_8 &= \text{gm1}\cdot X_9,\\
X_9 &= \text{fm1}\cdot X_{10}+\text{cm}\cdot X_{91},\\
X_{10} &= \text{1pa}\cdot X_{11},\\
X_{11} &= \text{ga1}\cdot X_{12},\\
X_{12} &= \text{fa1}\cdot X_{13}+\text{ca}\cdot X_{121},\ \text{and}\\
X_{13} &= \text{1a}\cdot \mathcal{M}^s[\![ABP]\!].
\end{aligned}\tag{3.7.19}$$

The meaning of the abbreviations X_1 through X_{13} in the above equations is as follows.

$$\begin{aligned}
X_1 &= \lambda_{[0ms]}(((\text{0rt}_1+\text{0rt}_2)^*\text{0a})\cdot S1\cdot(S^*\delta)\parallel MRA),\\
X_2 &= \lambda_{\mathbf{0}}(((\text{0rt}_1+\text{0rt}_2)^*\text{0a})\cdot S1\cdot(S^*\delta)\parallel(\text{fm0}+\text{cm})\cdot(M^*\delta)\parallel RA),\\
X_3 &= \lambda_{[0mr]}(((\text{0rt}_1+\text{0rt}_2)^*\text{0a})\cdot S1\cdot(S^*\delta)\parallel MRA),\\
X_4 &= \lambda_{[0as]}(((\text{0rt}_1+\text{0rt}_2)^*\text{0a})\cdot S1\cdot(S^*\delta)\parallel R1\cdot(R^*\delta)\parallel MA),\\
X_5 &= \lambda_{\mathbf{0}}(((\text{0rt}_1+\text{0rt}_2)^*\text{0a})\cdot S1\cdot(S^*\delta)\parallel R1\cdot(R^*\delta)\parallel(\text{fa0}+\text{ca})\cdot(A^*\delta)\parallel M^*\delta),\\
X_6 &= \lambda_{[0ar]}(((\text{0rt}_1+\text{0rt}_2)^*\text{0a})\cdot S1\cdot(S^*\delta)\parallel R1\cdot(R^*\delta)\parallel MA),\\
X_7 &= \lambda_{\mathbf{0}}(S1\cdot(S^*\delta)\parallel R1\cdot(R^*\delta)\parallel MA),\\
X_8 &= \lambda_{[1ms]}(((\text{1rt}_1+\text{1rt}_2)^*\text{1a})\cdot(S^*\delta)\parallel R1\cdot(R^*\delta)\parallel MA),\\
X_9 &= \lambda_{\mathbf{0}}(((\text{1rt}_1+\text{1rt}_2)^*\text{1a})\cdot(S^*\delta)\parallel(\text{fm1}+\text{cm})\cdot(M^*\delta)\parallel R1\cdot(R^*\delta)\parallel A^*\delta),
\end{aligned}\tag{3.7.20}$$

$$
\begin{aligned}
X_{10} &= \lambda_{[1mr]}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*1\mathrm{a}) \cdot (S^*\delta) \parallel R1 \cdot (R^*\delta) \parallel MA), \\
X_{11} &= \lambda_{[1as]}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*1\mathrm{a}) \cdot (S^*\delta) \parallel MRA), \\
X_{12} &= \lambda_{\mathbf{0}}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*1\mathrm{a}) \cdot (S^*\delta) \parallel (\mathrm{fa1} + \mathrm{ca}) \cdot (A^*\delta) \parallel MR), \text{ and} \\
X_{13} &= \lambda_{[1ar]}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*1\mathrm{a}) \cdot (S^*\delta) \parallel MRA).
\end{aligned} \tag{3.7.20 ctd.}
$$

Informally, the modular P/T net *ABP* first transmits a 0 message (action 0t), which is forwarded by the message channel in two steps (actions gm0 and fm0). Upon receiving the message, the receiver returns a positive acknowledgment to the sender (0pa), which is forwarded by the acknowledgment channel (ga0, fa0). After the sender has received the acknowledgment (0a), it sends the next message with a 1 bit (1t), provided of course that a message is available from the environment. The message is forwarded through the channel (gm1, fm1) and acknowledged by the receiver (1pa); the acknowledgment is forwarded by the channel (ga1, fa1) and received by the sender (1a). At this point, *ABP* has returned to its initial state characterized by $\mathcal{M}^s[\![ABP]\!]$.

As mentioned, at four points, it is possible that a message or acknowledgment is corrupted by a channel. These points are states X_2, X_5, X_9, and X_{12}. Consider the equation for X_2 in Result 3.7.19 on the preceding page. It states that the corruption of a 0 message leads to state X_{21}. If this happens, the protocol enters a small iteration described by the following equations. The equations for X_1 and X_2 are taken from Result 3.7.19.

$$
\begin{aligned}
X_2 &= \mathrm{fm0} \cdot X_3 + \mathrm{cm} \cdot X_{21}, \\
X_{21} &= 1\mathrm{na}_1 \cdot X_{22}, \\
X_{22} &= \mathrm{ga1} \cdot X_{23}, \\
X_{23} &= \mathrm{ca} \cdot X_{24} + \mathrm{fa1} \cdot X_{25}, \\
X_{24} &= 0\mathrm{rt}_1 \cdot X_1, \\
X_{25} &= 0\mathrm{rt}_2 \cdot X_1, \text{ and} \\
X_1 &= \mathrm{gm0} \cdot X_2.
\end{aligned} \tag{3.7.22}
$$

The abbreviations X_{21}, X_{22}, X_{23}, X_{24}, and X_{25} in the above equations denote the following terms. (See Definition 3.7.20 on the previous page for the meaning of X_1 and X_2.)

$$
\begin{aligned}
X_{21} &= \lambda_{[\perp m]}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2)^*0\mathrm{a}) \cdot S1 \cdot (S^*\delta) \parallel MRA), \\
X_{22} &= \lambda_{[1as]}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2)^*0\mathrm{a}) \cdot S1 \cdot (S^*\delta) \parallel MRA), \\
X_{23} &= \lambda_{\mathbf{0}}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2)^*0\mathrm{a}) \cdot S1 \cdot (S^*\delta) \parallel (\mathrm{fa1} + \mathrm{ca}) \cdot (A^*\delta) \parallel MR), \\
X_{24} &= \lambda_{[\perp a]}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2)^*0\mathrm{a}) \cdot S1 \cdot (S^*\delta) \parallel MRA), \text{ and} \\
X_{25} &= \lambda_{[1ar]}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2)^*0\mathrm{a}) \cdot S1 \cdot (S^*\delta) \parallel MRA).
\end{aligned} \tag{3.7.23}
$$

Informally, one cycle of this iteration does the following. The initial situation is that a 0 message is inside the message channel *MC* (state X_2). If the message is corrupted (action cm), the receiver sends an acknowledgment with bit 1 attached (action $1\mathrm{na}_1$), which is meant to be a *negative* acknowledgment indicating that the transmission of the 0 message has failed. The acknowledgment passes through the channel that may or may not corrupt it (actions ga1, ca, and fa1). If the sender receives an uncorrupted 1 acknowledgment, it knows that its attempt to transmit a 0 message has failed; if it receives a corrupted acknowledgment, it does not know whether the transmission has succeeded or not. In either

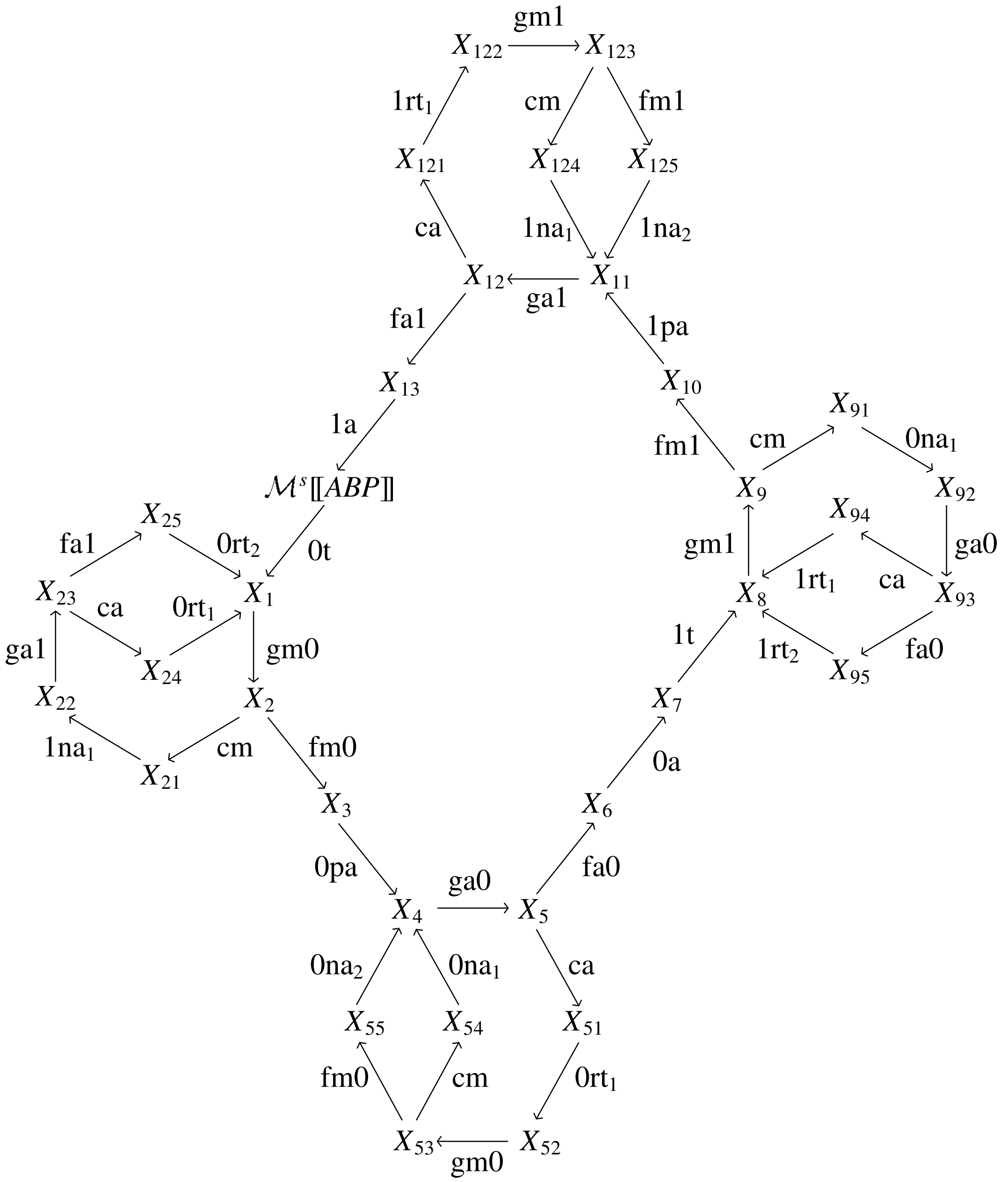


Figure 3.7.21: The system behavior of *ABP*.

case, the sender decides to retransmit the 0 message ($0\mathrm{rt}_1$, $0\mathrm{rt}_2$). When the message enters the channel (gm0), the protocol returns to the initial state of the iteration (state X_2).

It remains to be proven that the equations for X_2, X_{21}, X_{22}, X_{23}, X_{24}, X_{25}, and X_1 given in Result 3.7.22 on Page 100 indeed describe an iteration. Substitution yields the following equation for X_2.

$$X_2 = \mathrm{fm0} \cdot X_3 + \mathrm{cm} \cdot 1\mathrm{na}_1 \cdot \mathrm{ga1} \cdot (\mathrm{ca} \cdot 0\mathrm{rt}_1 + \mathrm{fa1} \cdot 0\mathrm{rt}_2) \cdot \mathrm{gm0} \cdot X_2. \tag{3.7.24}$$

It follows from RSP^* (and $A1$) that

$$X_2 = (\mathrm{cm} \cdot 1\mathrm{na}_1 \cdot \mathrm{ga1} \cdot (\mathrm{ca} \cdot 0\mathrm{rt}_1 + \mathrm{fa1} \cdot 0\mathrm{rt}_2) \cdot \mathrm{gm0}) * (\mathrm{fm0} \cdot X_3). \tag{3.7.25}$$

By means of $BKS2$, this last result can be rewritten into the following more convenient format.

$$X_2 = ((\mathrm{cm} \cdot 1\mathrm{na}_1 \cdot \mathrm{ga1} \cdot (\mathrm{ca} \cdot 0\mathrm{rt}_1 + \mathrm{fa1} \cdot 0\mathrm{rt}_2) \cdot \mathrm{gm0}) * \mathrm{fm0}) \cdot X_3. \tag{3.7.26}$$

The second point where the main loop of the protocol may be interrupted because a channel corrupts a message or an acknowledgment is state X_5 (see Result 3.7.19 on Page 99). In this case, the acknowledgment channel may corrupt the acknowledgment sent by the receiver to acknowledge the proper receipt of a 0 message (ca). If this happens, the sender does not know that its message has been received properly. So, it decides to retransmit its 0 message ($0\mathrm{rt}_1$). This message is either forwarded properly by the channel (gm0, fm0) or it may be corrupted (gm0, cm). In the first case, the receiver gets a duplicate of the 0 message. In reaction, it sends a 0 negative acknowledgment indicating that it has already received the 0 message. In the second case, the receiver does not know whether its own 0 acknowledgment or the sender's next 1 message got corrupted. In either case, it sends a 0 acknowledgment, indicating that it wants the sender to (re-)transmit a 1 message. The following equations formally describe the behavior sketched above. The equations for X_4 and X_5 are taken from Result 3.7.19 on Page 99.

$$\begin{aligned}
X_5 &= \mathrm{fa0} \cdot X_6 + \mathrm{ca} \cdot X_{51}, \\
X_{51} &= 0\mathrm{rt}_1 \cdot X_{52}, \\
X_{52} &= \mathrm{gm0} \cdot X_{53}, \\
X_{53} &= \mathrm{cm} \cdot X_{54} + \mathrm{fm0} \cdot X_{55}, \\
X_{54} &= 0\mathrm{na}_1 \cdot X_4, \\
X_{55} &= 0\mathrm{na}_2 \cdot X_4, \text{ and} \\
X_4 &= \mathrm{ga0} \cdot X_5.
\end{aligned} \tag{3.7.27}$$

The meaning of X_{51}, X_{52}, X_{53}, X_{54}, and X_{55} is as follows. (See Definition 3.7.20 on Page 99 for the meaning of X_4 and X_5.)

$$\begin{aligned}
X_{51} &= \lambda_{[\perp a]}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2) * 0\mathrm{a}) \cdot S1 \cdot (S * \delta) \parallel R1 \cdot (R * \delta) \parallel MA), \\
X_{52} &= \lambda_{[0ms]}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2) * 0\mathrm{a}) \cdot S1 \cdot (S * \delta) \parallel R1 \cdot (R * \delta) \parallel MA), \\
X_{53} &= \lambda_{\mathbf{0}}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2) * 0\mathrm{a}) \cdot S1 \cdot (S * \delta) \parallel \\
&\qquad\qquad (\mathrm{fm0} + \mathrm{cm}) \cdot (M * \delta) \parallel R1 \cdot (R * \delta) \parallel (A * \delta) \\
&\qquad), \\
X_{54} &= \lambda_{[\perp m]}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2) * 0\mathrm{a}) \cdot S1 \cdot (S * \delta) \parallel R1 \cdot (R * \delta) \parallel MA), \text{ and} \\
X_{55} &= \lambda_{[0mr]}(((0\mathrm{rt}_1 + 0\mathrm{rt}_2) * 0\mathrm{a}) \cdot S1 \cdot (S * \delta) \parallel R1 \cdot (R * \delta) \parallel MA).
\end{aligned} \tag{3.7.28}$$

By means of substitution and Axioms $A1$, RSP^*, and $BKS2$ it easily follows that

$$X_5 = ((\mathrm{ca}\cdot 0\mathrm{rt}_1\cdot \mathrm{gm0}\cdot(\mathrm{cm}\cdot 0\mathrm{na}_1 + \mathrm{fm0}\cdot 0\mathrm{na}_2)\cdot \mathrm{ga0})^*\,\mathrm{fa0})\cdot X_6. \qquad (3.7.29)$$

The other two times that the main iteration of the protocol may be interrupted because a message or acknowledgment is corrupted are the states X_9 and X_{12} (see Result 3.7.19 on Page 99). The resulting behavior in these states is very similar to the behavior in states X_2 and X_5, respectively. The only difference is that the roles of the 0 and 1 bit are interchanged. The results are summarized as follows. First,

$$\begin{aligned}
X_9 &= \mathrm{fm1}\cdot X_{10} + \mathrm{cm}\cdot X_{91},\\
X_{91} &= 0\mathrm{na}_1\cdot X_{92},\\
X_{92} &= \mathrm{ga0}\cdot X_{93},\\
X_{93} &= \mathrm{ca}\cdot X_{94} + \mathrm{fa0}\cdot X_{95},\\
X_{94} &= 1\mathrm{rt}_1\cdot X_8,\\
X_{95} &= 1\mathrm{rt}_2\cdot X_8, \text{ and}\\
X_8 &= \mathrm{gm1}\cdot X_9,
\end{aligned} \qquad (3.7.30)$$

with

$$\begin{aligned}
X_{91} &= \lambda_{[\perp m]}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*\,1\mathrm{a})\cdot(S^*\delta)\parallel R1\cdot(R^*\delta)\parallel MA),\\
X_{92} &= \lambda_{[0as]}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*\,1\mathrm{a})\cdot(S^*\delta)\parallel R1\cdot(R^*\delta)\parallel MA),\\
X_{93} &= \lambda_{\mathbf{0}}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*\,1\mathrm{a})\cdot(S^*\delta)\parallel R1\cdot(R^*\delta)\\
&\qquad \parallel(\mathrm{fa0}+\mathrm{ca})\cdot(A^*\delta)\parallel M^*\delta\\
&\qquad),\\
X_{94} &= \lambda_{[\perp a]}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*\,1\mathrm{a})\cdot(S^*\delta)\parallel R1\cdot(R^*\delta)\parallel MA), \text{ and}\\
X_{95} &= \lambda_{[0ar]}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*\,1\mathrm{a})\cdot(S^*\delta)\parallel R1\cdot(R^*\delta)\parallel MA).
\end{aligned} \qquad (3.7.31)$$

Consequently,

$$X_9 = ((\mathrm{cm}\cdot 0\mathrm{na}_1\cdot \mathrm{ga0}\cdot(\mathrm{ca}\cdot 1\mathrm{rt}_1 + \mathrm{fa0}\cdot 1\mathrm{rt}_2)\cdot \mathrm{gm1})^*\,\mathrm{fm1})\cdot X_{10}. \qquad (3.7.32)$$

Second,

$$\begin{aligned}
X_{12} &= \mathrm{fa1}\cdot X_{13} + \mathrm{ca}\cdot X_{121},\\
X_{121} &= 1\mathrm{rt}_1\cdot X_{122},\\
X_{122} &= \mathrm{gm1}\cdot X_{123},\\
X_{123} &= \mathrm{cm}\cdot X_{124} + \mathrm{fm1}\cdot X_{125},\\
X_{124} &= 1\mathrm{na}_1\cdot X_{11},\\
X_{125} &= 1\mathrm{na}_2\cdot X_{11}, \text{ and}\\
X_{11} &= \mathrm{ga1}\cdot X_{12},
\end{aligned} \qquad (3.7.33)$$

with

$$\begin{aligned}
X_{121} &= \lambda_{[\perp a]}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*\,1\mathrm{a})\cdot(S^*\delta)\parallel MRA),\\
X_{122} &= \lambda_{[1ms]}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*\,1\mathrm{a})\cdot(S^*\delta)\parallel MRA),\\
X_{123} &= \lambda_{\mathbf{0}}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*\,1\mathrm{a})\cdot(S^*\delta)\parallel(\mathrm{fm1}+\mathrm{cm})\cdot(M^*\delta)\parallel RA),\\
X_{124} &= \lambda_{[\perp m]}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*\,1\mathrm{a})\cdot(S^*\delta)\parallel MRA), \text{ and}\\
X_{125} &= \lambda_{[1mr]}(((1\mathrm{rt}_1 + 1\mathrm{rt}_2)^*\,1\mathrm{a})\cdot(S^*\delta)\parallel MRA).
\end{aligned} \qquad (3.7.34)$$

Consequently,

$$X_{12} = ((\mathrm{ca} \cdot 1\mathrm{rt}_1 \cdot \mathrm{gm1} \cdot (\mathrm{cm} \cdot 1\mathrm{na}_1 + \mathrm{fm1} \cdot 1\mathrm{na}_2) \cdot \mathrm{ga1})^* \mathrm{fa1}) \cdot X_{13}. \tag{3.7.35}$$

It remains to calculate an expression for the overall behavior of *ABP*. Summarizing the results of the second part of the verification so far, yields the following equations.

$$\begin{aligned}
\mathcal{M}^s[\![ABP]\!] &= 0\mathrm{t} \cdot \mathrm{gm0} \cdot X_2, \\
X_2 &= ((\mathrm{cm} \cdot 1\mathrm{na}_1 \cdot \mathrm{ga1} \cdot (\mathrm{ca} \cdot 0\mathrm{rt}_1 + \mathrm{fa1} \cdot 0\mathrm{rt}_2) \cdot \mathrm{gm0})^* \mathrm{fm0}) \cdot X_3, \\
X_3 &= 0\mathrm{pa} \cdot \mathrm{ga0} \cdot X_5, \\
X_5 &= ((\mathrm{ca} \cdot 0\mathrm{rt}_1 \cdot \mathrm{gm0} \cdot (\mathrm{cm} \cdot 0\mathrm{na}_1 + \mathrm{fm0} \cdot 0\mathrm{na}_2) \cdot \mathrm{ga0})^* \mathrm{fa0}) \cdot X_6, \\
X_6 &= 0\mathrm{a} \cdot X_7, \\
X_7 &= 1\mathrm{t} \cdot \mathrm{gm1} \cdot X_9, \\
X_9 &= ((\mathrm{cm} \cdot 0\mathrm{na}_1 \cdot \mathrm{ga0} \cdot (\mathrm{ca} \cdot 1\mathrm{rt}_1 + \mathrm{fa0} \cdot 1\mathrm{rt}_2) \cdot \mathrm{gm1})^* \mathrm{fm1}) \cdot X_{10}, \\
X_{10} &= 1\mathrm{pa} \cdot \mathrm{ga1} \cdot X_{12}, \\
X_{12} &= ((\mathrm{ca} \cdot 1\mathrm{rt}_1 \cdot \mathrm{gm1} \cdot (\mathrm{cm} \cdot 1\mathrm{na}_1 + \mathrm{fm1} \cdot 1\mathrm{na}_2) \cdot \mathrm{ga1})^* \mathrm{fa1}) \cdot X_{13}, \text{ and} \\
X_{13} &= 1\mathrm{a} \cdot \mathcal{M}^s[\![ABP]\!].
\end{aligned} \tag{3.7.36}$$

Hence, *RSP** yields

$$\begin{aligned}
\mathcal{M}^s[\![ABP]\!] = (\,& 0\mathrm{t} \cdot \mathrm{gm0} \cdot \\
& \quad ((\mathrm{cm} \cdot 1\mathrm{na}_1 \cdot \mathrm{ga1} \cdot (\mathrm{ca} \cdot 0\mathrm{rt}_1 + \mathrm{fa1} \cdot 0\mathrm{rt}_2) \cdot \mathrm{gm0})^* \mathrm{fm0}) \cdot \\
& 0\mathrm{pa} \cdot \mathrm{ga0} \cdot \\
& \quad ((\mathrm{ca} \cdot 0\mathrm{rt}_1 \cdot \mathrm{gm0} \cdot (\mathrm{cm} \cdot 0\mathrm{na}_1 + \mathrm{fm0} \cdot 0\mathrm{na}_2) \cdot \mathrm{ga0})^* \mathrm{fa0}) \cdot \\
& 0\mathrm{a} \cdot \\
& 1\mathrm{t} \cdot \mathrm{gm1} \cdot \\
& \quad ((\mathrm{cm} \cdot 0\mathrm{na}_1 \cdot \mathrm{ga0} \cdot (\mathrm{ca} \cdot 1\mathrm{rt}_1 + \mathrm{fa0} \cdot 1\mathrm{rt}_2) \cdot \mathrm{gm1})^* \mathrm{fm1}) \cdot \\
& 1\mathrm{pa} \cdot \mathrm{ga1} \cdot \\
& \quad ((\mathrm{ca} \cdot 1\mathrm{rt}_1 \cdot \mathrm{gm1} \cdot (\mathrm{cm} \cdot 1\mathrm{na}_1 + \mathrm{fm1} \cdot 1\mathrm{na}_2) \cdot \mathrm{ga1})^* \mathrm{fa1}) \cdot \\
& 1\mathrm{a} \\
&)^* \delta.
\end{aligned} \tag{3.7.37}$$

Result 3.7.37 completes the computation of the system behavior of *ABP* and, therefore, the second part of the verification.

Verification, Part 3. In the final part of the verification, the consumptions and productions that are internal to the modular net *ABP* are hidden. This means that an expression for the black-box behavior of *ABP* is calculated, which should satisfy the specification for *ABP* as explained on Page 95. Recall Result 3.7.7 on Page 94 that states the following.

$$\mathcal{M}^b[\![ABP]\!] = \tau_I(\mathcal{M}^s[\![ABP]\!]).$$

Set I, containing the internal places of *ABP*, is equal to $\{0ms, 1ms, 0mr, 1mr, \perp m, 0as, 1as, 0ar, 1ar, \perp a\}$. Using the abbreviations of the actions of *ABP* given in Table 3.7.16 on Page 97 and Axiom *TA*2, it is easy to verify that for all actions a of *ABP* *not* in {0t, 1t, 0pa, 1pa}, the abstraction $\tau_I(a)$ yields the silent action $(\mathbf{0}, \mathbf{0})$. For a in {0t, 1t}, $\tau_I(a)$ is equal to $([i], \mathbf{0})$; for a in {0pa, 1pa}, $\tau_I(a)$ is equal to $(\mathbf{0}, [o])$. In the verification below, the Fair

Iteration Rule (*FIR*) is used. This is allowed because the informal system requirements on Page 89 say that a channel will not always corrupt the same message. For the sake of readability, the silent action $(\mathbf{0},\mathbf{0})$ is written τ.

$$
\begin{aligned}
& \tau_I(\mathcal{M}^s[\![ABP]\!]) \\
= \quad & \{ \text{ Result 3.7.37 on the preceding page; Axioms } TA1\text{–}5 \ \} \\
& (\,([i],\mathbf{0})\cdot\tau\cdot \\
& \qquad ((\tau\cdot\tau\cdot\tau\cdot(\tau\cdot\tau+\tau\cdot\tau)\cdot\tau)^*\tau)\cdot \\
& \quad (\mathbf{0},[o])\cdot\tau\cdot \\
& \qquad ((\tau\cdot\tau\cdot\tau\cdot(\tau\cdot\tau+\tau\cdot\tau)\cdot\tau)^*\tau)\cdot \\
& \quad \tau\cdot \\
& \quad ([i],\mathbf{0})\cdot\tau\cdot \\
& \qquad ((\tau\cdot\tau\cdot\tau\cdot(\tau\cdot\tau+\tau\cdot\tau)\cdot\tau)^*\tau)\cdot \\
& \quad (\mathbf{0},[o])\cdot\tau\cdot \\
& \qquad ((\tau\cdot\tau\cdot\tau\cdot(\tau\cdot\tau+\tau\cdot\tau)\cdot\tau)^*\tau)\cdot \\
& \quad \tau \\
&)\,^*\delta \\
= \quad & \{ \ A3,\ B1 \text{ (repeatedly) } \} \\
& (([i],\mathbf{0})\cdot(\tau^*\tau)\cdot(\mathbf{0},[o])\cdot(\tau^*\tau)\cdot([i],\mathbf{0})\cdot(\tau^*\tau)\cdot(\mathbf{0},[o])\cdot(\tau^*\tau))^*\delta \\
= \quad & \{ \ FIR \text{ (repeatedly) } \} \\
& (([i],\mathbf{0})\cdot(\tau\cdot\tau+\tau)\cdot(\mathbf{0},[o])\cdot(\tau\cdot\tau+\tau)\cdot \\
& \qquad ([i],\mathbf{0})\cdot(\tau\cdot\tau+\tau)\cdot(\mathbf{0},[o])\cdot(\tau\cdot\tau+\tau) \\
&)^*\delta \\
= \quad & \{ \ B1,\ A3 \text{ (repeatedly) } \} \\
& (([i],\mathbf{0})\cdot(\mathbf{0},[o])\cdot([i],\mathbf{0})\cdot(\mathbf{0},[o]))^*\delta
\end{aligned}
$$

Summarizing,

$$\mathcal{M}^b[\![ABP]\!] = (([i],\mathbf{0})\cdot(\mathbf{0},[o])\cdot([i],\mathbf{0})\cdot(\mathbf{0},[o]))^*\delta. \tag{3.7.38}$$

This last expression is almost the desired result. Recall Property 2.4.64 on Page 57 stating that, for any guard p (see Definition 2.4.56 on Page 55), $p^*\delta = (p\cdot p)^*\delta$. Applying Property 2.4.64 to Result 3.7.38 yields that

$$\mathcal{M}^b[\![ABP]\!] = (([i],\mathbf{0})\cdot(\mathbf{0},[o]))^*\delta. \tag{3.7.39}$$

Result 3.7.11 on Page 95 implies that *ABP* indeed satisfies its specification.

$$\mathcal{M}^b[\![ABP]\!] = [\![ABP_{\text{spec}}]\!]. \tag{3.7.40}$$

This last result is equal to the proof requirement of Property 3.7.10 on Page 95. It completes the verification of the modular P/T net *ABP* of Figure 3.7.3 on Page 92. Note, however, that it remains to complete the design of the four components of *ABP*.

3.7.4 The P/T-net components of the modular net

The behavioral specifications of the components of *ABP* have already been given in Section 3.7.2. The next step in the design process is to construct P/T-net components according to these specifications.

Petri-net model. As already mentioned, one generic channel component is designed. The message and acknowledgment channels are instances of this generic channel. Figure 3.7.41 recalls the specification of the channel given in Definition 3.7.4 on Page 92 and it shows a corresponding P/T-net component. The interface of component *CH* has already been explained in Section 3.7.2. The meaning of the internal places and the transitions of the component is as follows. The P/T-net component of Figure 3.7.41 should be clear without further explanation.

$j \in \{0, 1\}$	
tj	: transfer for message j.
gj	: get message j.
fj	: forward message j.
cj	: corrupt message j.

Specification:

$$CH_{\text{spec}} = (i0? \cdot (o0! + \bot!) + i1? \cdot (o1! + \bot!))^* \delta$$

CH :

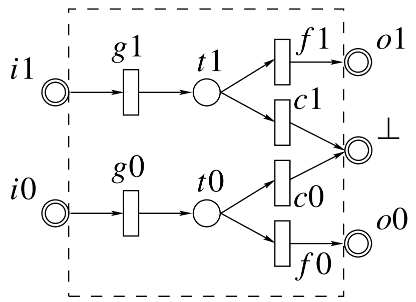


Figure 3.7.41: A P/T-net component modeling a channel.

Figure 3.7.42 on the next page shows the P/T-net components for the sender and receiver. It also recalls their specifications which are given in Definitions 3.7.5 and 3.7.6 in Section 3.7.2. The interfaces of *SN* and *RC* have also been explained in Section 3.7.2. The meaning of the internal places and the transitions is as follows.

$j \in \{0, 1\}$	
Sender	
js	: ready to send a j message.
jw	: waiting for a j acknowledgment.
jt	: transmit a j message.
jrt_1, jrt_2	: retransmit a j message.
ja	: acknowledge a j message.

$j \in \{0, 1\}$	
Receiver	
jr	: ready to receive a j message.
jpa	: acknowledge a j message.
jna_1, jna_2	: send a negative j acknowledgment.

Specifications:

$$SN_{\text{spec}} = (di? \mid 0mo! \cdot ((\perp ai? \mid 0mo! + 1ai? \mid 0mo!) \,{}^*\, 0ai?) \cdot \\ di? \mid 1mo! \cdot ((\perp ai? \mid 1mo! + 0ai? \mid 1mo!) \,{}^*\, 1ai?) \\ ) \,{}^*\, \delta$$

$$RC_{\text{spec}} = (((\perp mi? \mid 1ao! + 1mi? \mid 1ao!) \,{}^*\, 0mi? \mid 0ao! \mid do!) \cdot \\ ((\perp mi? \mid 0ao! + 0mi? \mid 0ao!) \,{}^*\, 1mi? \mid 1ao! \mid do!) \\ ) \,{}^*\, \delta$$

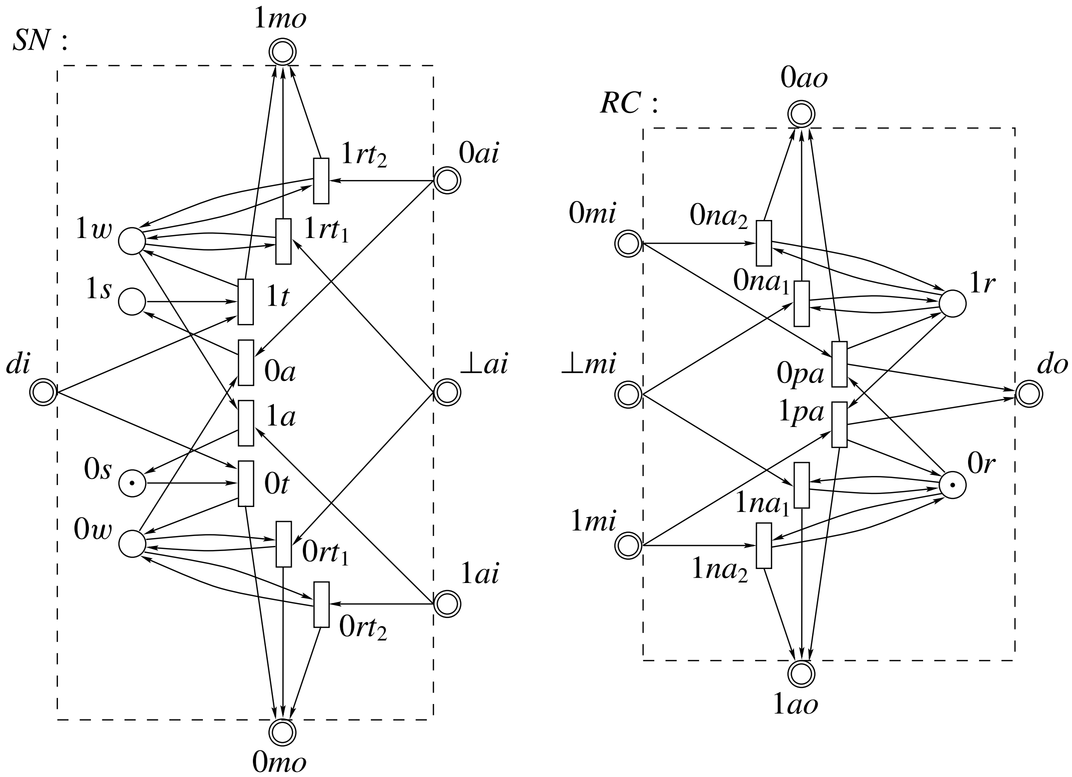


Figure 3.7.42: P/T-net components modeling the sender and receiver.

Simulation. At this point, the Petri-net model of the ABP is complete, which means that it can be simulated using tools as ExSpect and Design/CPN. However, it is beyond the scope of this thesis to discuss simulation of Petri nets in detail. Since the net model of the ABP is very simple, the simulation results are not very surprising.

3.7.5 Verifying the P/T-net components

The final step of the design process of the modular-net model of the ABP is the verification of the three components given in the previous subsection. This means that the following three equations have to be proven.

$$\begin{aligned}&\mathcal{C}^b[\![CH]\!] = [\![CH_{\mathrm{spec}}]\!],\\ &\mathcal{C}^b[\![SN]\!] = [\![SN_{\mathrm{spec}}]\!], \text{ and}\\ &\mathcal{C}^b[\![RC]\!] = [\![RC_{\mathrm{spec}}]\!].\end{aligned} \tag{3.7.43}$$

Verification of the channel. Definitions 3.7.1 (Interpretation $[\![\_]\!]$) on Page 90 and 3.7.4 (Specification CH_{spec}) on Page 92 yield the following result.

$$\begin{aligned}&[\![CH_{\mathrm{spec}}]\!] =\\ &\quad (([i0], \mathbf{0}) \cdot ((\mathbf{0}, [o0]) + (\mathbf{0}, [\bot])) + ([i1], \mathbf{0}) \cdot ((\mathbf{0}, [o1]) + (\mathbf{0}, [\bot])))^* \delta.\end{aligned} \tag{3.7.44}$$

Let I be the set $\{t0, t1\}$ of internal places of CH. The identifiers of transitions of component CH are used as abbreviations of the corresponding actions defined by the glass-box action function α^g (see Definition 3.3.9 on Page 64). Definition 3.4.15 (Algebraic black-box semantics) on Page 72 states that

$$\mathcal{C}^b[\![CH]\!] = \tau_I(\mathcal{C}^g[\![CH]\!]). \tag{3.7.45}$$

According to Definition 3.4.6 (Algebraic glass-box semantics) on Page 67,

$$\mathcal{C}^g[\![CH]\!] = \lambda^I_{\mathbf{0}}(g0^*\delta \parallel f0^*\delta \parallel c0^*\delta \parallel g1^*\delta \parallel f1^*\delta \parallel c1^*\delta). \tag{3.7.46}$$

The following result is not difficult to verify. The calculations are much simpler than the ones in Section 3.7.3 for the verification of the modular net *ABP*. The derivation for *CH* is very similar to the one in Example 3.4.10 on Page 70 for the simple production-unit example used in earlier sections of this chapter.

$$\mathcal{C}^g[\![CH]\!] = (g0 \cdot (f0 + c0) + g1 \cdot (f1 + c1)) \cdot \mathcal{C}^g[\![CH]\!]. \tag{3.7.47}$$

Hence, RSP^* yields that

$$\mathcal{C}^g[\![CH]\!] = (g0 \cdot (f0 + c0) + g1 \cdot (f1 + c1))^*\delta. \tag{3.7.48}$$

The final step is to hide internal consumptions and productions. It easily follows from the axioms of the abstraction operator, Axioms *TA*1–5, that

$$\tau_I(\mathcal{C}^g[\![CH]\!]) = [\![CH_{\mathrm{spec}}]\!]. \tag{3.7.49}$$

Hence, combining Results 3.7.45 and 3.7.49 means that the equation for channel *CH* in Property 3.7.43 is proven.

Verification of the sender. Definitions 3.7.1 (Interpretation $[\![\_]\!]$) on Page 90 and 3.7.5 (Specification SN_{spec}) on Page 93 yield that

$$
\begin{aligned}
&[\![SN_{\text{spec}}]\!] = \\
&\quad (\,([i],[0mo]) \cdot (\,(([\perp ai],[0mo]) + ([1ai],[0mo]))^{*}([0ai],\mathbf{0})\,) \cdot \\
&\qquad\quad ([i],[1mo]) \cdot (\,(([\perp ai],[1mo]) + ([0ai],[1mo]))^{*}([1ai],\mathbf{0})\,) \\
&\quad)^{*}\delta.
\end{aligned} \tag{3.7.50}
$$

Let I be the set $\{0s, 0w, 1s, 1w\}$ of internal places. As before, let identifiers of transitions be abbreviations of the corresponding glass-box actions. Let S be an abbreviation of the term $0t^{*}\delta \parallel 0a^{*}\delta \parallel 0rt_1{}^{*}\delta \parallel 0rt_2{}^{*}\delta \parallel 1t^{*}\delta \parallel 1a^{*}\delta \parallel 1rt_1{}^{*}\delta \parallel 1rt_2{}^{*}\delta$. Definitions 3.4.15 (Algebraic black-box semantics) on Page 72 and 3.4.6 (Algebraic glass-box semantics) on Page 67 yield that

$$\mathcal{C}^{b}[\![SN]\!] = \tau_I(\mathcal{C}^{g}[\![SN]\!]), \tag{3.7.51}$$

with

$$\mathcal{C}^{g}[\![SN]\!] = \lambda^{I}_{[0s]}(S). \tag{3.7.52}$$

It is not difficult to verify the following results. The superscript of the state operator is omitted.

$$
\begin{aligned}
\lambda_{[0s]}(S) &= 0t \cdot \lambda_{[0w]}(S), \\
\lambda_{[0w]}(S) &= 0a \cdot \lambda_{[1s]}(S) + 0rt_1 \cdot \lambda_{[0w]}(S) + 0rt_2 \cdot \lambda_{[0w]}(S), \\
\lambda_{[1s]}(S) &= 1t \cdot \lambda_{[1w]}(S), \text{ and} \\
\lambda_{[1w]}(S) &= 1a \cdot \lambda_{[0s]}(S) + 1rt_1 \cdot \lambda_{[1w]}(S) + 1rt_2 \cdot \lambda_{[1w]}(S).
\end{aligned} \tag{3.7.53}
$$

Applying RSP^{*} and $BKS2$ yields the following results.

$$
\begin{aligned}
\lambda_{[0w]}(S) &= ((0rt_1 + 0rt_2)^{*}0a) \cdot \lambda_{[1s]}(S) \text{ and} \\
\lambda_{[1w]}(S) &= ((1rt_1 + 1rt_2)^{*}1a) \cdot \lambda_{[0s]}(S)
\end{aligned} \tag{3.7.54}
$$

Combining Results 3.7.52, 3.7.53, and 3.7.54, and applying RSP^{*} one more time yields the following expression for the glass box behavior of the sender component.

$$\mathcal{C}^{g}[\![SN]\!] = (0t \cdot ((0rt_1 + 0rt_2)^{*}0a) \cdot 1t \cdot ((1rt_1 + 1rt_2)^{*}1a))^{*}\delta. \tag{3.7.55}$$

Finally, hiding the internal consumptions and productions easily yields the second equation of Property 3.7.43 on the preceding page.

$$\mathcal{C}^{b}[\![SN]\!] = \tau_I(\mathcal{C}^{g}[\![SN]\!]) = [\![SN_{\text{spec}}]\!]. \tag{3.7.56}$$

Verification of the receiver. The verification of the receiver is very similar to the verification of the sender. Let I be the set $\{0r, 1r\}$ of internal places. Definitions 3.4.15 (Algebraic black-box semantics) on Page 72 and 3.4.6 (Algebraic glass-box semantics) on Page 67 yield that

$$\mathcal{C}^{b}[\![RC]\!] = \tau_I(\mathcal{C}^{g}[\![RC]\!]), \tag{3.7.57}$$

with

$$\mathcal{C}^g[\![RC]\!] = \lambda^I_{[0r]}(0pa^*\delta \parallel 0na_1{}^*\delta \parallel 0na_2{}^*\delta \parallel 1pa^*\delta \parallel 1na_1{}^*\delta \parallel 1na_2{}^*\delta). \qquad (3.7.58)$$

The following result is straightforward.

$$\mathcal{C}^g[\![RC]\!] = (((1na_1 + 1na_2)^*0pa) \cdot ((0na_1 + 0na_2)^*1pa))^*\delta. \qquad (3.7.59)$$

Hiding the internal consumptions and productions in this last result yields the third equation of Property 3.7.43 on Page 108.

$$\mathcal{C}^b[\![RC]\!] = [\![RC_{\text{spec}}]\!]. \qquad (3.7.60)$$

Concluding remarks. Result 3.7.60 completes the design and analysis of the ABP. The case study shows that the design method presented in Section 3.2 is useful in structuring the design process. The combination of behavioral specification, Petri-net modeling, simulation, and algebraic verification improves the understanding of the system under development. Moreover, the method guarantees that the system satisfies its behavioral specification. This is particularly useful for concurrent systems of which the communication behavior is critical. A drawback of the theory is that the algebraic verification of modular P/T nets is tedious. It remains unclear from the verification of the ABP whether the method scales to real-world concurrent systems.

3.7.6 A remark on the equivalence of modular P/T nets

So far, the theory developed in this chapter has been used to verify P/T nets against their specifications. However, it can also be used in a slightly different way. Given a modular P/T net or a P/T-net component, its algebraic black-box semantics can be used to calculate the behavior that is observable to the environment. This means that it is possible to determine whether two nets have the same observable behavior. For many purposes, such as simulation and some analysis techniques, nets with equivalent behavior may be interchanged.

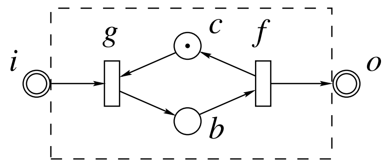


Figure 3.7.61: A P/T-net component modeling a one-place buffer.

Consider, for example, P/T-net component B depicted in Figure 3.7.61 which models a one-place buffer. Its interface consists of an input place i and an output place o. It has a capacity which is determined by the number of tokens in c. In the case of a one-place buffer, the capacity is of course one. Items that are put into the buffer are stored in place

b. It is not difficult to verify that net component B satisfies the specification of the ABP given in Definition 3.7.2 on Page 90. That is,

$$\mathcal{C}^b[\![B]\!] = (([i], \mathbf{0}) \cdot (\mathbf{0}, [o]))^* \delta = [\![ABP_{\text{spec}}]\!]. \tag{3.7.62}$$

Assume a scenario where a designer is building a large Petri-net model of a distributed system with components of the system communicating via the Alternating-Bit Protocol. The designer wants to simulate and analyze the distributed system. He or she is not interested in properties of the ABP, because the net-model of the ABP, for example the modular net of Figure 3.7.3 on Page 92, has already been proven correct. In this case, it can be convenient to replace all instances of *ABP* by the one-place buffer of Figure 3.7.61 on the preceding page. Doing so will improve the efficiency of simulations. In addition, it may be helpful to simplify analysis of the system by means of, for example, invariant analysis and state-space analysis (see, among others, (PETERSON, 1981; REISIG, 1985; MURATA, 1989) and Section 2.3). If used as sketched in this example, the theory developed in this chapter may be a further aid in the compositional design and analysis of concurrent systems.

3.8 Case Study: The Production Unit

3.8.1 Introduction

To validate the method developed in this chapter, it is necessary to show that it can be applied to real-world design problems. Therefore, in this section, the method is applied to the development of a production unit. This case study is a simplified version of a problem that originated from Dutch industry. The design problem is more complex than the design of the ABP done in the previous section. In particular, it includes concurrency, whereas the ABP is a strictly sequential process. On the one hand, the production-unit case study does indeed serve as a further validation of the method of this chapter. On the other hand, it clearly shows the limitations of the underlying formalism, pointing out possibilities for improvements and future work.

Informal system requirements. The informal description of the production unit and its required behavior is as follows. The most simple version of the unit, called a *basic unit*, can perform a single operation on a single piece of unprocessed material. First, it must receive a command. Then, it requests and receives material, performs the operation specified in the command, and waits for an output request. Upon receiving an output request, it delivers the processed material. Typically, the processing of material takes the most time in the whole system. While the unit is processing material, it is ready to receive the next command. In this way, the delay between two operations in a unit is minimized. As soon as the unit is ready to receive a new command, it sends a request to its environment.

It must be possible to combine several basic units into so-called *complex* units by connecting them in series. For this purpose, a *generic* control component must be developed that controls *two* units. These two units are not necessarily basic ones. They can be complex units built themselves from basic units and controllers. In this way, it must be possible to construct arbitrarily long series of basic units, using only two components, namely the basic unit and the generic controller. Therefore, it is essential that the interface behavior of a series of basic units is similar to the behavior of a single basic unit.

Especially the last requirement is rather vague, but, as often in practice, that is more or less the way the problem was phrased. The case study is a simplified version of the real case, because it may be assumed that no errors can occur during the processing of the material in a basic unit. In reality, this assumption is not true. Furthermore, in the real case, it was also possible to combine two units in parallel, raising the need for a second controller component.

3.8.2 The basic production unit

Behavioral specification. From the informal system requirements in the previous section, the following actions of a basic unit can be derived. They are specified in the simple specification language introduced in Section 3.7.1. This means that they should be interpreted as consumptions and productions of tokens in a Petri-net model.

$cmd?$	: receive a command.
$irq!$	: request input material.
$imt?$	: receive input material.
$ncrq!$	: send a new-command request.
$orq?$	: receive an output request.
$omt!$	: deliver output material.

The behavior of a basic unit can now be specified as follows.

$$BUnit_{\text{spec}} = cmd? \cdot ((irq! \cdot imt? \cdot (ncrq! \cdot cmd? \parallel orq? \mid omt!))^* \delta). \tag{3.8.1}$$

This specification deserves a few words of explanation. It is not difficult to see that the basic unit first receives a command after which it enters a non-terminating repetition. Each cycle of the repetition starts with an input request followed by the receipt of material. Upon receiving material, the unit starts its processing task, which does not result in a visible action. It continues by sending a new-command request followed by the receipt of the next command and *in parallel* it may deliver the output material, provided of course that the processing is completed. After it has delivered the output *and* it has received a new command, it starts with the next cycle of its iteration. In the above specification, the unit refuses to receive an output request if the output material is not available. Only if both the request and the material are available, the request is processed and the material delivered. A slightly different, but equally correct specification is to let the unit first receive the output request and then deliver the material.

Petri-net model. In the actual case study, a colored-Petri-net model of the basic unit was developed, using the tool ExSpect. However, for the purpose of this section, details about data types and timing are omitted. The focus is on the modular construction and the communication structure instead.

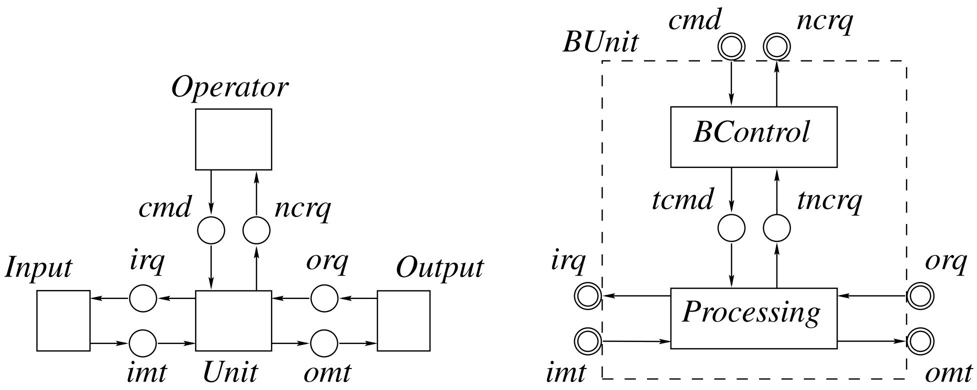


Figure 3.8.2: A basic production unit and its environment.

Figure 3.8.2 shows a modular P/T net of the basic unit and its environment. The environment consists of an input system, an output system, and an operator. It is the same for both basic and complex units. The environment is simply given for the sake of clarity. No details of the input system, the output system, and the operator are given. The basic unit is divided into two separate subsystems, a control system, *BControl*, and a processing system, *Processing*. The control system transfers commands and new-command requests between the environment and the processing system. The processing system, obviously, takes care of the main processing task.

Behavioral specification of the P/T-net components. The next step is to give specifications for the components of *BUnit*. The pins of *BControl* are as follows. Mapping f_c maps pins of *BControl* to places of *BUnit*.

		f_c
cicmd	: input command	*cmd*
cocmd	: output command	*tcmd*
cincrq	: input new-command request	*tncrq*
concrq	: output new-command request	*ncrq*

Specification $BUnit_{\text{spec}}$ of the unit shows that the receipt of a command and the sending of a new-command request alternate. Given the informal requirement that *BControl* transfers commands and new-command requests between the environment and the processing system, the following specification for *BControl* makes sense.

$$BControl_{\text{spec}} = (cicmd? \mid cocmd! \cdot cincrq? \mid concrq!)^* \delta. \tag{3.8.3}$$

The processing part is specified as follows. As before, actions are processed simultaneously as much as possible. Upon receiving a command, an input request is sent; upon receiving input material, a new command is requested and upon receiving an output request, the output material is delivered. Mapping f_p maps pins of *Processing* to places of *BUnit*.

		f_p
pcmd	: input command	*tcmd*
pncrq	: output new-command request	*tncrq*
pirq	: input request	*irq*
porq	: output request	*orq*
pimt	: input material	*imt*
pomt	: output material	*omt*

$$Processing_{\text{spec}} = (pcmd? \mid pirq! \cdot pimt? \mid pncrq! \cdot porq? \mid pomt!)^* \delta. \tag{3.8.4}$$

It may be surprising that neither *BControl* nor *Processing* contain explicit parallelism. However, if one has a close look at the specifications and the modular net *BUnit* of Figure 3.8.2 on the preceding page, one can see that *Processing* produces a token in place *tncrq* at the same time it receives input material. Hence, the transfer of the new-command request to the environment by component *BControl* can occur in parallel with the delivery of output material by component *Processing*. This is exactly the desired result. The informal argument of this paragraph is confirmed by the verification in the next step of the design cycle.

Verification. Given the specifications for *BControl* and *Processing*, it is possible to verify the behavior of the basic unit. The following property must be proven.

$$\mathcal{M}^b[\![BUnit]\!] = [\![BUnit_{\text{spec}}]\!]. \tag{3.8.5}$$

Let I be the set $\{tcmd, tncrq\}$ of internal places of *BUnit*. Let $\boldsymbol{c}$ and $\boldsymbol{p}$ be the renaming functions on multi-actions defined in terms of the partial place-fusion functions f_c and f_p given above. Definitions 3.6.7 (Algebraic black-box semantics) on Page 88 and 3.6.4 (Algebraic system semantics) on Page 87 yield that

$$\mathcal{M}^b[\![BUnit]\!] = \tau_I(\mathcal{M}^s[\![BUnit]\!]), \tag{3.8.6}$$

with

$$\mathcal{M}^s[\![BUnit]\!] = \lambda^I_{\mathbf{0}}(\rho_{\boldsymbol{c}}(\mathcal{C}^b[\![BControl]\!]) \parallel \rho_{\boldsymbol{p}}(\mathcal{C}^b[\![Processing]\!])). \tag{3.8.7}$$

Since it may be assumed that *BControl* and *Processing* satisfy their specifications, the following equalities may be used.

$$\mathcal{C}^b[\![BControl]\!] = [\![BControl_{\text{spec}}]\!] \text{ and} \tag{3.8.8}$$
$$\mathcal{C}^b[\![Processing]\!] = [\![Processing_{\text{spec}}]\!].$$

The following table shows some abbreviations that are used in the verification.

Control		
$([cmd],[tcmd])$	rcmd	receive a command.
$([tncrq],[ncrq])$	sncrq	send a new-command request.
Processing		
$([tcmd],[irq])$	rqmat	request material.
$([imt],[tncrq])$	imat	input material.
$([orq],[omt])$	omat	output material.

As in the previous section, where the Alternating-Bit Protocol has been verified, the verification of the modular net consists of three parts. In the first part, expressions are determined for the behavior of the components in the context of the modular net. In the second part, the system behavior of the modular net is calculated. In the third part, internal consumptions and productions are hidden. The results of the first part of the verification are easily verified.

$$\begin{aligned}&\rho_{\boldsymbol{c}}(\mathcal{C}^b[\![BControl]\!]) = (\mathrm{rcmd}\cdot\mathrm{sncrq})\,{}^*\,\delta \text{ and}\\&\rho_{\boldsymbol{p}}(\mathcal{C}^b[\![Processing]\!]) = (\mathrm{rqmat}\cdot\mathrm{imat}\cdot\mathrm{omat})\,{}^*\,\delta.\end{aligned} \tag{3.8.9}$$

In the second part of the verification, an expression is calculated for the system behavior of *BUnit*. Let C and P be abbreviations of $\rho_{\boldsymbol{c}}(\mathcal{C}^b[\![BControl]\!])$ and $\rho_{\boldsymbol{p}}(\mathcal{C}^b[\![Processing]\!])$, respectively. The following equations summarize some straightforward results. The superscript I of the state operator, equal to $\{tcmd, tncrq\}$, is omitted.

$$\begin{aligned}&\lambda_{\mathbf{0}}(C \parallel P) = \mathrm{rcmd}\cdot\lambda_{[tcmd]}(\mathrm{sncrq}\cdot C \parallel P),\\&\lambda_{[tcmd]}(\mathrm{sncrq}\cdot C \parallel P) = \mathrm{rqmat}\cdot\mathrm{imat}\cdot\lambda_{[tncrq]}(\mathrm{sncrq}\cdot C \parallel \mathrm{omat}\cdot P),\\&\lambda_{[tncrq]}(\mathrm{sncrq}\cdot C \parallel \mathrm{omat}\cdot P) =\\&\qquad\quad \mathrm{sncrq}\cdot\lambda_{\mathbf{0}}(C \parallel \mathrm{omat}\cdot P) + \mathrm{omat}\cdot\lambda_{[tncrq]}(\mathrm{sncrq}\cdot C \parallel P),\\&\lambda_{\mathbf{0}}(C \parallel \mathrm{omat}\cdot P) = \mathrm{rcmd}\cdot\mathrm{omat}\cdot\lambda_{[tcmd]}(\mathrm{sncrq}\cdot C \parallel P) + \mathrm{omat}\cdot\lambda_{\mathbf{0}}(C \parallel P), \text{ and}\\&\lambda_{[tncrq]}(\mathrm{sncrq}\cdot C \parallel P) = \mathrm{sncrq}\cdot\lambda_{\mathbf{0}}(C \parallel P).\end{aligned} \tag{3.8.10}$$

The equations of Result 3.8.10 can be combined to derive a result for $\lambda_{[tcmd]}(\mathrm{sncrq}\cdot C \parallel P)$. The reason for considering this expression instead of any of the others is that specification $BUnit_{\mathrm{spec}}$ given in Definition 3.8.1 on Page 112 shows that $\lambda_{[tcmd]}(\mathrm{sncrq}\cdot C \parallel P)$ should be the beginning of an iteration.

$$\begin{aligned}&\quad\lambda_{[tcmd]}(\mathrm{sncrq}\cdot C \parallel P)\\=&\qquad \{\text{ Result 3.8.10; repeated substitution }\}\\&\quad\mathrm{rqmat}\cdot\mathrm{imat}\cdot\ (\ \mathrm{sncrq}\cdot\ (\ \mathrm{rcmd}\cdot\mathrm{omat}\cdot\lambda_{[tcmd]}(\mathrm{sncrq}\cdot C \parallel P)\\&\qquad\qquad\qquad\qquad\qquad + \mathrm{omat}\cdot\mathrm{rcmd}\cdot\lambda_{[tcmd]}(\mathrm{sncrq}\cdot C \parallel P)\\&\qquad\qquad\qquad\qquad\qquad)\\&\qquad\qquad\qquad + \mathrm{omat}\cdot\mathrm{sncrq}\cdot\mathrm{rcmd}\cdot\lambda_{[tcmd]}(\mathrm{sncrq}\cdot C \parallel P)\\&\qquad\qquad\qquad)\\=&\qquad \{\ A4(2\times)\ \}\\&\quad\mathrm{rqmat}\cdot\mathrm{imat}\cdot(\mathrm{sncrq}\cdot(\mathrm{rcmd}\cdot\mathrm{omat} + \mathrm{omat}\cdot\mathrm{rcmd}) + \mathrm{omat}\cdot\mathrm{sncrq}\cdot\mathrm{rcmd})\,\cdot\\&\qquad\qquad\lambda_{[tcmd]}(\mathrm{sncrq}\cdot C \parallel P)\end{aligned}$$

Applying *RSP** yields the expected iteration.

$$\lambda_{[tcmd]}(\text{sncrq} \cdot C \parallel P) = \\ (\text{rqmat} \cdot \text{imat} \cdot (\text{sncrq} \cdot (\text{rcmd} \cdot \text{omat} + \text{omat} \cdot \text{rcmd}) + \text{omat} \cdot \text{sncrq} \cdot \text{rcmd}))^{\,*}\delta. \tag{3.8.11}$$

At this point, two approaches are possible. It follows from specification $BUnit_{\text{spec}}$ in Definition 3.8.1 on Page 112 that several actions in the iteration occur in parallel. The above result does not contain any concurrency. In one approach, this concurrent behavior is reconstructed. In the other approach, this reconstruction is omitted. Both approaches have advantages and disadvantages. Therefore, in the remainder, they are both explained.

The first approach is to reconstruct the parallelism. As mentioned, it can be derived from specification $BUnit_{\text{spec}}$ in Definition 3.8.1 on Page 112 which actions should occur in parallel. Consider the following derivation.

$$\begin{array}{ll}
& \text{sncrq} \cdot (\text{rcmd} \cdot \text{omat} + \text{omat} \cdot \text{rcmd}) + \text{omat} \cdot \text{sncrq} \cdot \text{rcmd} \\
= & \quad \{\, M2 \text{ (repeatedly)} \,\} \\
& \text{sncrq} \cdot (\text{rcmd} \mathbin{⫣} \text{omat} + \text{omat} \mathbin{⫣} \text{rcmd}) + \text{omat} \mathbin{⫣} \text{sncrq} \cdot \text{rcmd} \\
= & \quad \{\, M1 \,\} \\
& \text{sncrq} \cdot (\text{rcmd} \parallel \text{omat}) + \text{omat} \mathbin{⫣} \text{sncrq} \cdot \text{rcmd} \\
= & \quad \{\, M3 \,\} \\
& \text{sncrq} \cdot \text{rcmd} \mathbin{⫣} \text{omat} + \text{omat} \mathbin{⫣} \text{sncrq} \cdot \text{rcmd} \\
= & \quad \{\, M1 \,\} \\
& \text{sncrq} \cdot \text{rcmd} \parallel \text{omat}
\end{array}$$

Combining this result with Result 3.8.11 yields that

$$\lambda_{[tcmd]}(\text{sncrq} \cdot C \parallel P) = (\text{rqmat} \cdot \text{imat} \cdot (\text{sncrq} \cdot \text{rcmd} \parallel \text{omat}))^{\,*}\delta. \tag{3.8.12}$$

It follows from Result 3.8.7 and the first equation of Result 3.8.10 on the previous page that

$$\mathcal{M}^s[\![BUnit]\!] = \text{rcmd} \cdot \lambda_{[tcmd]}(\text{sncrq} \cdot C \parallel P). \tag{3.8.13}$$

Hence, Results 3.8.12 and 3.8.13 yield the following expression for the system behavior of the basic unit.

$$\mathcal{M}^s[\![BUnit]\!] = \text{rcmd} \cdot ((\text{rqmat} \cdot \text{imat} \cdot (\text{sncrq} \cdot \text{rcmd} \parallel \text{omat}))^{\,*}\delta). \tag{3.8.14}$$

The third part of the verification in the first approach consists of hiding the internal consumptions and productions. Let I be equal to $\{tcmd, tncrq\}$.

$$\begin{array}{ll}
& \tau_I(\mathcal{M}^s[\![BUnit]\!]) \\
= & \quad \{\, \text{Substitution} \,\} \\
& \tau_I(\text{rcmd} \cdot ((\text{rqmat} \cdot \text{imat} \cdot (\text{sncrq} \cdot \text{rcmd} \parallel \text{omat}))^{\,*}\delta)) \\
= & \quad \{\, TA1{,}4{,}5{,}TAM1 \,\} \\
& \tau_I(\text{rcmd}) \cdot ((\tau_I(\text{rqmat}) \cdot \tau_I(\text{imat}) \cdot (\tau_I(\text{sncrq}) \cdot \tau_I(\text{rcmd}) \parallel \tau_I(\text{omat})))^{\,*}\delta) \\
= & \quad \{\, TA2 \,\} \\
& ([cmd], \mathbf{0}) \cdot (((\mathbf{0}, [irq]) \cdot ([imt], \mathbf{0}) \cdot ((\mathbf{0}, [ncrq]) \cdot ([cmd], \mathbf{0}) \parallel ([orq], [omt])))^{\,*}\delta)
\end{array}$$

Hence, it follows from Result 3.8.6 on Page 114 that

$$\mathcal{M}^b[\![BUnit]\!] = \tau_I(\mathcal{M}^s[\![BUnit]\!]) = \\ ([cmd], \mathbf{0}) \cdot (((\mathbf{0}, [irq]) \cdot ([imt], \mathbf{0}) \cdot ((\mathbf{0}, [ncrq]) \cdot ([cmd], \mathbf{0}) \parallel ([orq], [omt])))^* \delta). \tag{3.8.15}$$

Definitions 3.7.1 (Interpretation $[\![\_]\!]$) on Page 90 and 3.8.1 (Specification $BUnit_{\mathrm{spec}}$) on Page 112 give the following algebraic expression for $BUnit_{\mathrm{spec}}$.

$$[\![BUnit_{\mathrm{spec}}]\!] = \\ ([cmd], \mathbf{0}) \cdot (((\mathbf{0}, [irq]) \cdot ([imt], \mathbf{0}) \cdot ((\mathbf{0}, [ncrq]) \cdot ([cmd], \mathbf{0}) \parallel ([orq], [omt])))^* \delta). \tag{3.8.16}$$

Combining the last two results immediately yields the proof requirement of Property 3.8.5 on Page 114.

$$\mathcal{M}^b[\![BUnit]\!] = [\![BUnit_{\mathrm{spec}}]\!]. \tag{3.8.17}$$

The interesting aspect of the verifications in the first approach is that they contain concurrency. However, the approach has two disadvantages. First, it is up to the designer to recognize and reconstruct concurrent behavior. As the derivation above shows, the algebraic axioms are applied from right to left, which is opposite to the usual direction. For complex systems, the reconstruction of concurrency is often very tedious, even with the help of the specification of the modular net. Second, it is also difficult to use tools to support this approach. Tools for process-algebraic reasoning are often based on rewriting techniques. Such tools have difficulties handling verifications in which axioms are applied in both directions. An advantage of the approach is that the result for the system behavior is a compact and intuitive expression. In addition, hiding the internal consumptions and productions of the modular net is straightforward. In conclusion, the approach sketched above is mainly suitable for proofs by hand and for relatively small systems with only little concurrency.

In the second approach, the reconstruction of concurrent behavior in Result 3.8.11 on the preceding page is omitted. It is not difficult to derive the following expression for the system semantics of the basic unit (see Results 3.8.11 and 3.8.13).

$$\mathcal{M}^s[\![BUnit]\!] = \mathrm{rcmd} \cdot \\ ((\mathrm{rqmat} \cdot \mathrm{imat} \cdot (\mathrm{sncrq} \cdot (\mathrm{rcmd} \cdot \mathrm{omat} + \mathrm{omat} \cdot \mathrm{rcmd}) + \mathrm{omat} \cdot \mathrm{sncrq} \cdot \mathrm{rcmd}))^* \delta). \tag{3.8.18}$$

As before, the final step of the verification is to hide internal consumptions and productions. Recall that I is equal to $\{tcmd, tncrq\}$. The axioms for the abstraction operator yield the following result.

$$\begin{aligned} &\tau_I(\mathcal{M}^s[\![BUnit]\!]) = \\ &\quad ([cmd], \mathbf{0}) \cdot ((\,(\mathbf{0}, [irq]) \cdot ([imt], \mathbf{0}) \cdot \\ &\qquad\qquad ((\mathbf{0}, [ncrq]) \cdot \\ &\qquad\qquad\quad (([cmd], \mathbf{0}) \cdot ([orq], [omt]) + ([orq], [omt]) \cdot ([cmd], \mathbf{0})) \\ &\qquad\qquad + ([orq], [omt]) \cdot (\mathbf{0}, [ncrq]) \cdot ([cmd], \mathbf{0}) \\ &\qquad\qquad) \\ &\qquad)^* \delta \\ &\quad). \end{aligned} \tag{3.8.19}$$

Instead of reconstructing the concurrent behavior in the above result, the parallelism in the specification of the basic unit is eliminated. Recall Result 3.8.16 on the preceding page. The parallel behavior can be eliminated from the expression for the specification of *BUnit* by means of a few applications of the axioms for the merge operator. Essentially, the derivation on Page 116 is repeated in reversed order. The result is an expression identical to the one given in Result 3.8.19 on the previous page for the black-box behavior of *BUnit*. Consequently, the same result is obtained as in the first approach (Result 3.8.17 on the preceding page), satisfying the proof requirement of Property 3.8.5 on Page 114.

A disadvantage of the second approach is that the expression for the system behavior is much larger and less intuitive than the expression derived in the first approach. Consequently, the proof is less readable than in the first approach. A clear advantage of the second approach is that it is more suitable for tool support and, hence, more suitable for complex systems.

No matter what approach is taken, the verifications in this section show one of the main weaknesses of the theory. The axiomatization of the causal state operator is such that one is forced to compute every single action of the Petri net that is being verified. The result is always an algebraic term containing only choices and sequential compositions. Essentially, this term is nothing but the operational semantics of the P/T net. Since non-trivial concurrent systems often contain thousands of states, in practice, it is impossible to use the design method developed in this chapter without proper tool support.

Petri-net model of the components. It remains to complete the design of the components of the basic unit. Figure 3.8.20 on the facing page shows two P/T-net components modeling the control and processing parts of the unit. The meaning of the internal places and the transitions is as follows.

BControl	
rdy	: ready to receive a command.
bsy	: busy processing a command.
rcmd	: receive a command.
sncrq	: send a new-command request.
Processing	
empty	: no material present.
wfmt	: waiting for material.
rmt	: raw material.
pmt	: processed material.
rqmat	: request material.
imat	: input material.
pmat	: process material.
omat	: output material.

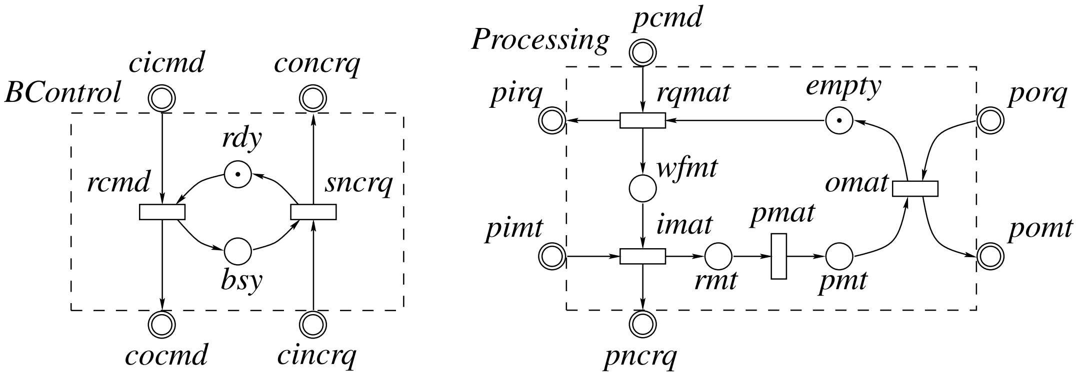


Figure 3.8.20: The control and processing components of the basic unit.

Verification of the P/T-net components. To prove the correctness of the components, the following two equalities must be proven.

$$\begin{aligned}&\mathcal{C}^b[\![BControl]\!] = [\![BControl_{\text{spec}}]\!] \text{ and}\\ &\mathcal{C}^b[\![Processing]\!] = [\![Processing_{\text{spec}}]\!].\end{aligned} \tag{3.8.21}$$

The calculations are standard and, therefore, omitted. Note that component *BControl* is identical – up to renaming of places and transitions – to the control component shown in Figure 3.5.2 on Page 76.

Properties of a basic unit. At this point, it is time to return to the requirement that the behavior of a series of basic units must be similar to the behavior of a single basic unit. It must be determined what are the fundamental properties in the dynamic behavior of a basic unit. The most obvious one is related to the interface between the unit and the operator. As mentioned earlier, the receipt of a command and the sending of a new-command request alternate. It is possible to prove this property formally. Hiding all consumptions and productions of the basic unit except the consumption of a *cmd* token and the production of a *ncrq* token shows that the unit satisfies behavior *CN*, which is defined as follows.

$$CN = (([cmd], \mathbf{0}) \cdot (\mathbf{0}, [ncrq]))^* \delta. \tag{3.8.22}$$

The calculations to prove that *BUnit* satisfies behavior *CN* are as follows. Let I be the set $\{irq, imt, orq, omt\}$. The silent action $(\mathbf{0}, \mathbf{0})$ is written τ.

$$\begin{aligned}
&\tau_I(\mathcal{M}^b[\![BUnit]\!])\\
=\quad &\{\text{ Result 3.8.19 on Page 117; Axioms } TA1\text{–}5 \}\\
&([cmd], \mathbf{0}) \cdot ((\tau \cdot \tau \cdot ((\mathbf{0}, [ncrq]) \cdot (([cmd], \mathbf{0}) \cdot \tau + \tau \cdot ([cmd], \mathbf{0}))\\
&\qquad\qquad\qquad + \tau \cdot (\mathbf{0}, [ncrq]) \cdot ([cmd], \mathbf{0})\\
&\qquad\qquad\qquad)\\
&\qquad\quad)^* \delta\\
&\qquad)\\
=\quad &\{ B1(2\times) \}
\end{aligned}$$

$$
\begin{aligned}
&([cmd],\mathbf{0})\cdot((\tau\cdot\ ((\mathbf{0},[ncrq])\cdot(([cmd],\mathbf{0})+\tau\cdot([cmd],\mathbf{0}))\\
&\qquad\qquad\qquad +\tau\cdot(\mathbf{0},[ncrq])\cdot([cmd],\mathbf{0})\\
&\qquad\qquad\qquad)\\
&\qquad\quad)^{*}\delta\\
&\qquad)\\
=&\quad \{\ A6,\ B2,\ A6\ \}\\
&([cmd],\mathbf{0})\cdot((\tau\cdot((\mathbf{0},[ncrq])\cdot([cmd],\mathbf{0})+\ \tau\cdot(\mathbf{0},[ncrq])\cdot([cmd],\mathbf{0})))^{*}\delta)\\
=&\quad \{\ A6,\ B2,\ A6\ \}\\
&([cmd],\mathbf{0})\cdot((\tau\cdot(\mathbf{0},[ncrq])\cdot([cmd],\mathbf{0}))^{*}\delta)
\end{aligned}
$$

Summarizing,

$$\tau_I(\mathcal{M}^b[\![BUnit]\!])=([cmd],\mathbf{0})\cdot((\tau\cdot(\mathbf{0},[ncrq])\cdot([cmd],\mathbf{0}))^{*}\delta). \tag{3.8.23}$$

This is almost the desired result. For the final step in the proof, an auxiliary property is needed. Let p and q be closed $\mathrm{PA}^{\tau*}_{\delta\lambda}+\mathrm{RN}$ terms such that $p\cdot q$ is a guard (see Definition 2.4.56 on Page 55).

$$
\begin{aligned}
&p\cdot((\tau\cdot q\cdot p)^{*}\delta)\\
=&\quad \{\ BKS1,\ A6\ \}\\
&p\cdot((\tau\cdot q\cdot p)\cdot((\tau\cdot q\cdot p)^{*}\delta))\\
=&\quad \{\ A5\text{ (repeatedly)},\ B1\ \}\\
&(p\cdot q)\cdot(p\cdot((\tau\cdot q\cdot p)^{*}\delta))
\end{aligned}
$$

Hence, RSP^{*} yields that

$$p\cdot((\tau\cdot q\cdot p)^{*}\delta)=(p\cdot q)^{*}\delta. \tag{3.8.24}$$

Applying Property 3.8.24 to Result 3.8.23 yields the final result.

$$\tau_I(\mathcal{M}^b[\![BUnit]\!])=(([cmd],\mathbf{0})\cdot(\mathbf{0},[ncrq]))^{*}\delta=CN. \tag{3.8.25}$$

Another characteristic property of the basic unit is that, looking at the stream of material, it behaves as a one-place buffer. Let I be the set $\{cmd, ncrq, irq, orq\}$. It is possible to prove the following property.

$$\tau_I(\mathcal{M}^b[\![BUnit]\!])=\tau\cdot((([imt],\mathbf{0})\cdot(\mathbf{0},[omt]))^{*}\delta). \tag{3.8.26}$$

Property 3.8.26 states that the basic unit after initialization, expressed by the silent action τ, enters a cycle in which it alternates the input of material and the output of material. The silent action represents the fact that the unit must receive a command before it can receive input. As explained in Example 2.2.18 on Page 12, initial silent actions cannot be removed in a semantics based on rooted branching bisimilarity. To prove Property 3.8.26, an auxiliary result, Property 3.8.27 given below, is needed. Using this property, the proof is not difficult. The details are left to the reader.

Let p and q be closed $\mathrm{PA}^{\tau*}_{\delta\lambda}+\mathrm{RN}$ terms such that p is a guard (see Definition 2.4.56 on Page 55).

$$
\begin{aligned}
&q\cdot((\tau\cdot p)^{*}\delta)\\
=&\quad \{\ BKS1,\ A6\ \}
\end{aligned}
$$

$$
\begin{array}{ll}
& q \cdot ((\tau \cdot p) \cdot ((\tau \cdot p) * \delta)) \\
= & \quad \{ A5\,(2\times), B1 \} \\
& q \cdot (p \cdot ((\tau \cdot p) * \delta)) \\
= & \quad \{ \text{Property 3.8.28 (below)} \} \\
& q \cdot (p * \delta)
\end{array}
$$

Hence,

$$q \cdot ((\tau \cdot p) * \delta) = q \cdot (p * \delta). \tag{3.8.27}$$

The proof of Property 3.8.27 uses the following result.

$$p \cdot ((\tau \cdot p) * \delta) = p * \delta, \tag{3.8.28}$$

which is simply proven by applying RSP^* to the result of the following derivation.

$$
\begin{array}{ll}
& p \cdot ((\tau \cdot p) * \delta) \\
= & \quad \{ BKS1, A6 \} \\
& p \cdot ((\tau \cdot p) \cdot ((\tau \cdot p) * \delta)) \\
= & \quad \{ A5\,(2\times), B1 \} \\
& p \cdot (p \cdot ((\tau \cdot p) * \delta))
\end{array}
$$

A final interesting property is that a basic unit can accept two commands before it must produce output. That is, it behaves as a two-place buffer with respect to commands. Let I be the set $\{ncrq, irq, imt, orq\}$. Using Property 3.8.27, it is straightforward to prove the following result.

$$\tau_I(\mathcal{M}^b[\![BUnit]\!]) = ([cmd], \mathbf{0}) \cdot ((([cmd], \mathbf{0}) \parallel (\mathbf{0}, [omt])) * \delta). \tag{3.8.29}$$

It is not difficult to see that the resulting behavior is indeed a two-place buffer. Property 3.8.29 shows that the unit satisfies the requirement that it must be able to receive a new command while it is still busy processing.

In conclusion, Properties 3.8.25, 3.8.26, and 3.8.29 characterize some essential aspects of the interface behavior of a basic unit. They conform to the informal system requirements given on Page 111. In the next subsection, the three properties are used as a guidance to specify some behavioral requirements for complex units.

3.8.3 Complex production units

Behavioral specification and Petri-net model. As explained in Section 3.8.1, it must be possible to put basic units in series of arbitrary length. Such series of basic units are called complex units. By means of complex units, it is possible to perform a series of operations on one piece of input material. The basis of complex units is a generic control component for two units in series. These two units may be basic units, but also complex units which are themselves constructed by means of the control component.

As already mentioned, the above description is all that is known about the requirements for complex units. The problem is how to translate these requirements into a behavioral specification. Assume that the complex unit under development consists of two units called $Unit_1$ and $Unit_2$. The first unit consists of a series of K_1 basic units, for some natural number K_1 at least one; the second unit consists of a series of K_2 basic units, where K_2 is also a natural number at least one. This means that the complex unit being designed consists of K basic units, where $K = K_1 + K_2$.

So far, always a complete specification of the behavior of the system under development has been given. However, in this case, the system is more complex than before and less is known about its desired behavior. On the other hand, based on the properties of a basic unit given in the previous section, it is relatively straightforward to derive some properties of complex units. In addition, it is not difficult to construct a modular P/T net of a complex unit. Therefore, instead of giving a complete specification of the behavior of complex units, a different approach is chosen. The starting point of the design process is a modular P/T net and a set of behavioral properties that the net model must satisfy.

The requirement that it must be possible to use a complex unit in the construction of other complex units means that the interface of complex units must be identical to the interface of basic units. Figure 3.8.30 shows a modular P/T net called *KUnit* which models a complex unit consisting of K basic units as described above. The idea is that a command received by *KUnit* contains information for both $Unit_1$ and $Unit_2$. Component *Control* forwards the command via places cmd_1 and cmd_2 to a unit as soon as it has received a new-command request from the unit via one of the places $ncrq_1$ or $ncrq_2$. The input request of $Unit_2$ connected to place *trq* serves as the output request of $Unit_1$. Material is transfered between $Unit_1$ and $Unit_2$ via place *tmt*.

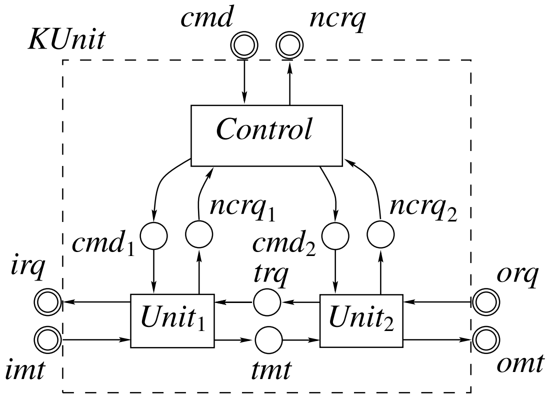


Figure 3.8.30: The modular net of a complex unit.

Note that the Petri net of Figure 3.8.30 is *not* a modular P/T net when Definition 3.5.1 on Page 75 is interpreted strictly. Even if $Unit_1$ and $Unit_2$ in Figure 3.8.30 are basic units as depicted in Figure 3.8.2 on Page 113, the net models of $Unit_1$ and $Unit_2$ are *modular P/T nets* themselves. Definition 3.5.1 does not allow modular P/T nets as components of another modular P/T net. However, if $Unit_1$ and $Unit_2$ are replaced by their unfolding, the result is a modular net in the strict sense of Definition 3.5.1. Recall that Theorem 3.5.14 on Page 80 shows that a modular P/T net and its unfolding have the same operational semantics. This means that it is allowed to generalize modular P/T nets to P/T nets with multiple hierarchical levels, as is done in Figure 3.8.30.

The behavioral properties that the modular P/T net *KUnit* must satisfy are generalizations of Properties 3.8.25, 3.8.26, and 3.8.29 for basic units derived in the previous subsection. First, consider Property 3.8.25 on Page 120, stating that a basic unit alternates commands and new-command requests. In order to be compatible with a basic unit, it is clear that a complex unit must also satisfy this property. Second, consider Property 3.8.26 on Page 120. This property states that a basic unit behaves as a one-place buffer when looking at the stream of material. Clearly, unit *KUnit* which consists of K basic units in series should behave as a K-place buffer. Finally, Property 3.8.29 on Page 121 states that a basic unit must be able to receive a second command before it produces an output. That is, it behaves as a two-place buffer with respect to commands. This property shows that a basic unit does not have to wait for commands unnecessarily. Since the second command in each basic unit is only stored and not actually processed, it is sufficient to require that *KUnit* behaves as a $(K+1)$-place buffer with respect to commands. In this way, no basic unit in the series has to wait unnecessarily.

In an ideal situation, the behavioral specification and verification of the modular net *KUnit* would be performed for arbitrary K. However, the current theory is not yet developed far enough to allow parameterized verifications. To allow such verifications, it is necessary to incorporate data in the theory of this chapter. An example of a process algebra with data types is μCRL (GROOTE AND PONSE, 1994). The process algebra μCRL contains induction principles that would allow us to verify the behavior of a complex unit for arbitrary K. The reason that the theory in this chapter has not yet been extended with data types and induction principles is that one important problem is still unsolved. It is not yet clear how to axiomatize the interaction between the causal state operator and the sum operator of μCRL. The sum operator of μCRL allows the specification of choices over infinite data domains. This operator is crucial in the verification of properties concerning, for example, natural numbers.

Since it is not clear how to include data types and induction principles in the theory, the goal to verify the behavior of the complex unit for arbitrary K is abandoned. Instead, the aim becomes to verify the behavior for a concrete value of K. The verification of the modular net $2Unit$, meaning that K equals two and the two units $Unit_1$ and $Unit_2$ are both basic units, seems to be a reasonable starting point.

The first step in the verification of $2Unit$ is to precisely specify the three behavioral requirements explained above. The expected behaviors are specified using the simple specification language introduced in Section 3.7.1. The alternation of commands and new-command requests is specified as follows.

$$CN_{\text{spec}} = (cmd? \cdot ncrq!)^{*}\,\delta. \tag{3.8.31}$$

Modular net $2Unit$ must behave as a two-place buffer with respect to the stream of material. Similar to the basic unit (see Property 3.8.26 on Page 120), $2Unit$ starts with performing some internal activity. The following specification defines the required input/output buffer behavior.

$$IO_{\text{spec}} = \tau \cdot imt? \cdot ((imt? \parallel omt!)^{*}\,\delta). \tag{3.8.32}$$

The final requirement, concerning the number of commands that $2\mathit{Unit}$ must be able to receive before any output is produced, is specified as follows. The requirement is defined as a *set* of specifications corresponding to a three-place buffer. It is not possible to define a three-place buffer by a single specification.

$$\begin{aligned}
CM_{\text{spec}} &= cmd? \cdot CM_1, \\
CM_1 &= cmd? \cdot CM_2 + omt! \cdot CM_{\text{spec}}, \\
CM_2 &= cmd? \cdot CM_3 + omt! \cdot CM_1, \text{ and} \\
CM_3 &= omt! \cdot CM_2.
\end{aligned} \tag{3.8.33}$$

Behavioral specification of component *Control*. Before it is possible to analyze the behavior of the modular net $2\mathit{Unit}$ of Figure 3.8.30 on Page 122, the behavior of the control component must be specified.

The interface of the P/T-net component *Control* contains the following pins. The mapping f_c connects pins of *Control* to places of the modular net $2\mathit{Unit}$.

$j \in \{1, 2\}$		f_c
$ccmd$	: input commands	cmd
$ccmd_j$	: output commands to unit j	cmd_j
$cncrq$	: new-command requests to the environment	$ncrq$
$cncrq_j$	: new-command requests from unit j	$ncrq_j$

The behavior of the generic control component is now specified as follows.

$$\begin{aligned}
Control_{\text{spec}} = {} & ccmd? \mid ccmd_1! \cdot \\
& (((ccmd_2! \parallel cncrq_1? \mid cncrq!) \cdot (ccmd? \mid ccmd_1! \parallel cncrq_2?))^* \delta).
\end{aligned} \tag{3.8.34}$$

This specification deserves a few words of explanation. Assume the production unit containing the control component has been running for some time. Furthermore, assume that the control component is in a state where it has forwarded the last command it received to the first unit, but not yet to the second unit. It has not yet received a new-command request from the first unit. The second unit is ready to receive a new command. Note that the command expected by the second unit is the one that the first unit is already processing.

In terms of the above specification, the control component is at the beginning of its main iteration. The main iteration consists of two phases. In the first phase, it can do two things in parallel. First, it can forward the command to the second unit. Second, it can receive a new-command request from the first unit. The new-command request is immediately forwarded to the environment to minimize delays. If both actions are completed, it enters the second phase of its main iteration. At the beginning of this second phase, it is ready to receive a new command. In addition, it may expect a new-command request from the second unit. These two actions can occur in parallel. If a command is received, it is forwarded immediately to the first unit, because this unit has already indicated that it is ready to receive a command. In addition, it is stored internally so that it can be forwarded to the second unit when it is ready. If a new-command request from the second unit is received, this indicates that the unit has started processing and is ready to receive its next command. The control component does not have to forward the new-command request to the environment, because it has already requested a new command in the first phase of the iteration. If both a new command and a new-command request from the second unit have been received, the control component is again in the initial state of the iteration as described above.

It remains to explain how the component arrives in the initial state of its iteration. The first time that it is in this state is when the first command has been received and forwarded to the first unit. This explains the initial action in the above specification.

At this point, the next steps would normally be the simulation and verification of the modular net $2Unit$. However, as explained in Section 3.7.2, simulation of incomplete modular nets as $2Unit$ is not possible with current tools. Nevertheless, the complexity of the design problem is such that simulation would be a helpful tool in testing the specification of the control component. The verification of the complex unit is a tedious task. It would be a waste of effort if any mistakes are overlooked.

In order to use simulation, it is necessary to complete the modular net $2Unit$. Two out of three components, namely the units, have already been designed and verified. This means that, to complete $2Unit$, it is only necessary to design a P/T-net component modeling the generic control component. Compared to the verification of the behavior of $2Unit$, this is a relatively straightforward task.

Therefore, the following approach is chosen. First, a net model of the control component is defined. Second, the complete modular net is simulated to test both the behavior of the control component and the behavior of the complete complex unit. Third, the control component is verified against its specification. It is useful to perform this step before the verification of the modular net $2Unit$ to guarantee that the simulation results are reliable. Obviously, they are not reliable if the verification shows that the control component used in the simulation does not satisfy its specification. Finally, the behavior of the complete unit is verified.

Petri-net model of component *Control*. Figure 3.8.35 on the following page shows a P/T-net component modeling the generic control component. The meaning of the internal places and the transitions is as follows.

$j \in \{1, 2\}$	
cw	: a command waiting for unit 2.
ncw	: no command is waiting.
rdy_j	: unit j is ready to receive a command.
bsy_j	: unit j is busy.
rn_j	: the control component is expecting to receive a new-command request from unit j.
$rcmd$	: receive a command and forward it to unit 1.
$fcmd$	: forward a command to unit 2.
$pncrq_j$	: process a new-command request from unit j.

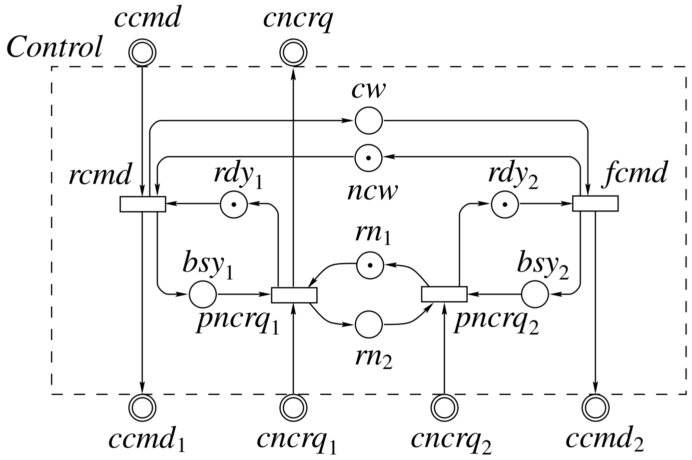


Figure 3.8.35: The control component of a complex unit.

Given the informal explanation of the specification of the control component in the previous paragraph, the net of Figure 3.8.35 should not be too hard to understand. The internal places of the component can be divided into four pairs. The pair *cw* and *ncw* guarantees that the firing of transitions *rcmd* and *fcmd* alternates. The token in *ncw* means that *rcmd* is the first transition to be fired. For j in $\{1, 2\}$, the pairs rdy_j and bsy_j make sure that forwarding a command to unit j and receiving a new-command request from unit j alternate. Finally, the pair rn_1 and rn_2 plays a similar role for transitions $pncrq_1$ and $pncrq_2$. It is easy to check that these properties conform to specification $Control_{\mathrm{spec}}$ (see Definition 3.8.34 on Page 124).

Simulation. It has already been explained that simulation of Petri nets is beyond the scope of this thesis. However, it should be mentioned that simulation proved to be a useful aid in the design process for complex production units. A colored version of the modular net 2*Unit* has been implemented and simulated in the tool ExSpect. Figure 3.8.36 on the next page shows a snapshot of a simulation session in ExSpect.

The two center windows show the ExSpect models of 2*Unit* and component *Control*. The marking of the control component shows that both basic units are busy processing. The control component has a command available for the second unit and it is waiting for its new-command request. The clock in the upper-left corner shows that the simulation time is 2.3 seconds. The other four windows each show a list of tokens for one particular place. Such a list contains all tokens that have been or still are present in the place. The list for `imt` shows that the unit has received three pieces of input material; the last piece of material is still being processed in `Unit1`. The list for `omt` shows that it has only produced one output. This means that the input/output behavior is at least a two-place buffer, which is consistent with the specification in Definition 3.8.32 on Page 123. The list for `cmd1` shows that the unit has received four commands. The last one is still available in place `cmd1`, waiting to be received by `Unit1`. Since the unit has only produced a single

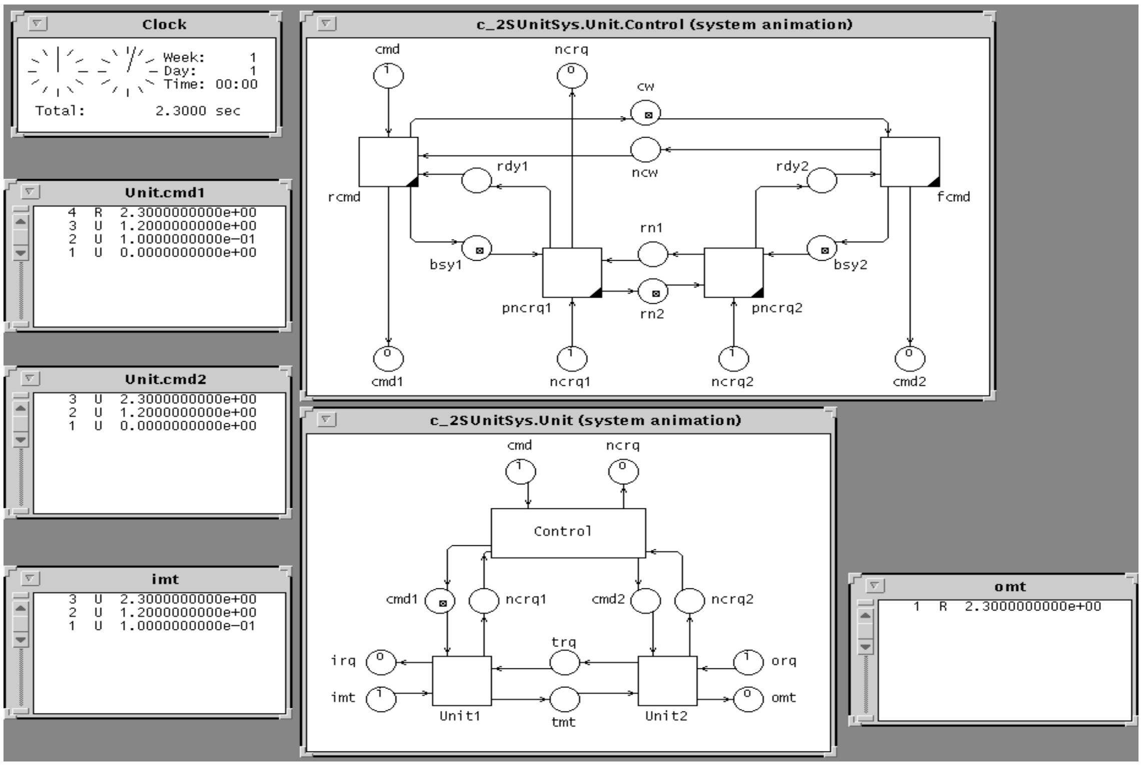


Figure 3.8.36: A snapshot of a simulation session.

output so far, this state is consistent with specification CM_{spec} (see Definition 3.8.33 on Page 124). When continuing the simulation, the first external action of the complex unit is an output action, as may be expected from the specifications in Definitions 3.8.32 and 3.8.33.

The brief explanation of a simulation session given in this paragraph shows only a fraction of the possibilities of the tool ExSpect to support the design, simulation, and analysis of (colored) Petri nets. A good introduction to ExSpect showing its application to a non-trivial application can be found in (Basten, Bol, and Voorhoeve, 1995). The reader interested in more details is referred to (ASPT, 1994).

In conclusion, the simulation sessions for the production unit were helpful in optimizing the design and correcting small mistakes in the early design process. They did not provide any results contradicting the behavioral requirements of the modular net 2*Unit* as specified in Definitions 3.8.31, 3.8.32, and 3.8.33. In addition to simulations for 2*Unit*, also some simulations of three units in series were done, testing 2*Unit* and the generic component *Control* in the context of a larger complex unit. The results of these simulation sessions also conformed to the specifications.

Verification of component *Control*. The next step is to verify the control component against its specification $Control_{\text{spec}}$, given in Definition 3.8.34 on Page 124. The complexity of the verification of the control component is about the same as the complexity of the verifications of other P/T-net components earlier in this chapter. Hence, the calculations are omitted. It suffices to say that component *Control* satisfies its specification.

$$\mathcal{C}^b[\![\mathit{Control}]\!] = [\![\mathit{Control}_{\mathrm{spec}}]\!] = ([\mathit{ccmd}], [\mathit{ccmd}_1]) \cdot \\ ((((\mathbf{0}, [\mathit{ccmd}_2]) \parallel ([\mathit{cncrq}_1], [\mathit{cncrq}])) \cdot (([\mathit{ccmd}], [\mathit{ccmd}_1]) \parallel ([\mathit{cncrq}_2], \mathbf{0}))) * \delta). \tag{3.8.37}$$

Verification of modular net $2\mathit{Unit}$. The final step of the design process is the verification of the behavior of the modular net $2\mathit{Unit}$. Modular net $2\mathit{Unit}$ is an instantiation of the modular net *KUnit* shown in Figure 3.8.30 on Page 122 where both Unit_1 and Unit_2 are basic units. The goal is to show that $2\mathit{Unit}$ satisfies the behavioral requirements given in Definitions 3.8.31 and 3.8.32 on Page 123 and 3.8.33 on Page 124. For this purpose, first, the black-box behavior $\mathcal{M}^b[\![2\mathit{Unit}]\!]$ is calculated. Second, by hiding the appropriate consumptions and productions, the three properties are verified. The computation of the black-box behavior is as usual separated into three parts.

The algebraic representation of the black-box behavior of $2\mathit{Unit}$ is as follows. Let I be the set of internal places of $2\mathit{Unit}$ (see Figure 3.8.30 on Page 122). Let f_c, f_o, and f_t be the mappings that connect pins of *Control*, Unit_1, and Unit_2 to places of $2\mathit{Unit}$, respectively. They can be easily derived from Figures 3.8.35 on Page 126, 3.8.2 on Page 113, and 3.8.30 on Page 122. Let $\boldsymbol{c}$, $\boldsymbol{o}$, and $\boldsymbol{t}$ be renaming functions on multi-actions defined in terms of mappings f_c, f_o, and f_t.

$$\mathcal{M}^b[\![2\mathit{Unit}]\!] = \tau_I(\mathcal{M}^s[\![2\mathit{Unit}]\!]) = \\ \tau_I(\lambda^I_{\mathbf{0}}(\rho_{\boldsymbol{c}}(\mathcal{C}^b[\![\mathit{Control}]\!]) \parallel \rho_{\boldsymbol{o}}(\mathcal{M}^b[\![\mathit{Unit}_1]\!]) \parallel \rho_{\boldsymbol{t}}(\mathcal{M}^b[\![\mathit{Unit}_2]\!]))). \tag{3.8.38}$$

As explained on Page 122, Unit_1 and Unit_2 are modular nets. Hence, their black-box behavior in the above expression is given by the semantic function $\mathcal{M}^b[\![\_]\!]$. It should be clear that this generalization of Definition 3.6.4 (Algebraic system semantics) on Page 87 is sound.

Verification, Part 1. In the first step of the verification, the behavior of the components of $2\mathit{Unit}$ in the context of the modular net is determined. That is, consumptions and productions of components are renamed according to the place-fusion function. The calculations are straightforward and, therefore, omitted. Recall that Result 3.8.37 contains an algebraic expression for $\mathcal{C}^b[\![\mathit{Control}]\!]$, the black-box behavior of component *Control*. Since Unit_1 and Unit_2 are basic units, their black-box behavior, $\mathcal{M}^b[\![\mathit{Unit}_1]\!]$ and $\mathcal{M}^b[\![\mathit{Unit}_2]\!]$, is defined by the expression in Result 3.8.15 on Page 117. The results of the first part of the verification are as follows. Table 3.8.40 on the next page explains the abbreviations of actions used in the expressions.

$$\rho_{\boldsymbol{c}}(\mathcal{C}^b[\![\mathit{Control}]\!]) = \mathrm{rcmd} \cdot (((\mathrm{fcmd} \parallel \mathrm{pncrq}_1) \cdot (\mathrm{rcmd} \parallel \mathrm{pncrq}_2)) * \delta), \\ \rho_{\boldsymbol{o}}(\mathcal{M}^b[\![\mathit{Unit}_1]\!]) = \mathrm{rcmd}_1 \cdot ((\mathrm{rqmat}_1 \cdot \mathrm{imat}_1 \cdot (\mathrm{sncrq}_1 \cdot \mathrm{rcmd}_1 \parallel \mathrm{omat}_1)) * \delta), \text{ and} \\ \rho_{\boldsymbol{t}}(\mathcal{M}^b[\![\mathit{Unit}_2]\!]) = \mathrm{rcmd}_2 \cdot ((\mathrm{rqmat}_2 \cdot \mathrm{imat}_2 \cdot (\mathrm{sncrq}_2 \cdot \mathrm{rcmd}_2 \parallel \mathrm{omat}_2)) * \delta). \tag{3.8.39}$$

Control		
$([cmd], [cmd_1])$	rcmd	receive a command and forward it to unit 1.
$(\mathbf{0}, [cmd_2])$	fcmd	forward a command to unit 2.
$([ncrq_1], [ncrq])$	pncrq_1	process a new-command request from unit 1.
$([ncrq_2], \mathbf{0})$	pncrq_2	process a new-command request from unit 2.
Units		
$j \in \{1, 2\}$		
$([cmd_j], \mathbf{0})$	rcmd_j	unit j receives a command.
$(\mathbf{0}, [ncrq_j])$	sncrq_j	unit j sends a new-command request.
$(\mathbf{0}, [irq])$	rqmat_1	unit 1 requests material.
$(\mathbf{0}, [trq])$	rqmat_2	unit 2 requests material.
$([imt], \mathbf{0})$	imat_1	unit 1 receives material.
$([tmt], \mathbf{0})$	imat_2	unit 2 receives material.
$([trq], [tmt])$	omat_1	unit 1 outputs material.
$([orq], [omt])$	omat_2	unit 2 outputs material.

Table 3.8.40: Abbreviations of actions of the modular net $2Unit$.

Verification, Part 2. In the second part of the verification, the system behavior of $2Unit$, $\mathcal{M}^s[\![2Unit]\!]$, is calculated. As the calculations for the ABP in Section 3.7.3 show, this part of the verification is tedious and may take many pages of calculations. However, once one is familiar with the theory, the basic steps are not very complicated. Therefore, the calculations are not given in detail. Instead, Figure 3.8.41 on the following page shows the operational semantics of $\mathcal{M}^s[\![2Unit]\!]$.

The process has 89 states numbered 0 through 88. In the remainder, these states are referred to as X_0 through X_{88}. Recall that process states in the operational semantics of algebraic terms are terms themselves. The initial state X_0 is equal to $\mathcal{M}^s[\![2Unit]\!]$.

For the sake of clarity, most labels of transitions have been omitted. The figure should be interpreted in such a way that opposite transitions in a diamond or a cube all carry the same label. For example, transitions (X_1, X_2), (X_3, X_5), (X_6, X_9), and (X_{10}, X_{14}) are all labeled rcmd_1. In the bottom part of the figure, all downward arrows are labeled omat_2.

Process $\mathcal{M}^s[\![2Unit]\!]$ is a recursive process. In Figure 3.8.41, process states with the same number should be identified. The dashed part of the figure is a repetition of the initial part of the process.

As already mentioned, the computation of the actions of $2Unit$ is not the most difficult part of the verification. A typical intermediate result of the verification is as follows. Let C, U_1, and U_2 be abbreviations for the expressions characterizing the main loops of the control component, unit one, and unit two, respectively (see also the expressions in Result 3.8.39 on the preceding page).

$$\begin{aligned} C &= ((\mathrm{fcmd} \parallel \mathrm{pncrq}_1) \cdot (\mathrm{rcmd} \parallel \mathrm{pncrq}_2))^*\delta \text{ and} \\ U_j &= (\mathrm{rqmat}_j \cdot \mathrm{imat}_j \cdot (\mathrm{sncrq}_j \cdot \mathrm{rcmd}_j \parallel \mathrm{omat}_j))^*\delta, \text{ for } j \in \{1, 2\}. \end{aligned} \tag{3.8.42}$$

Using these abbreviations, the following expression describing, for example, state X_{46} can

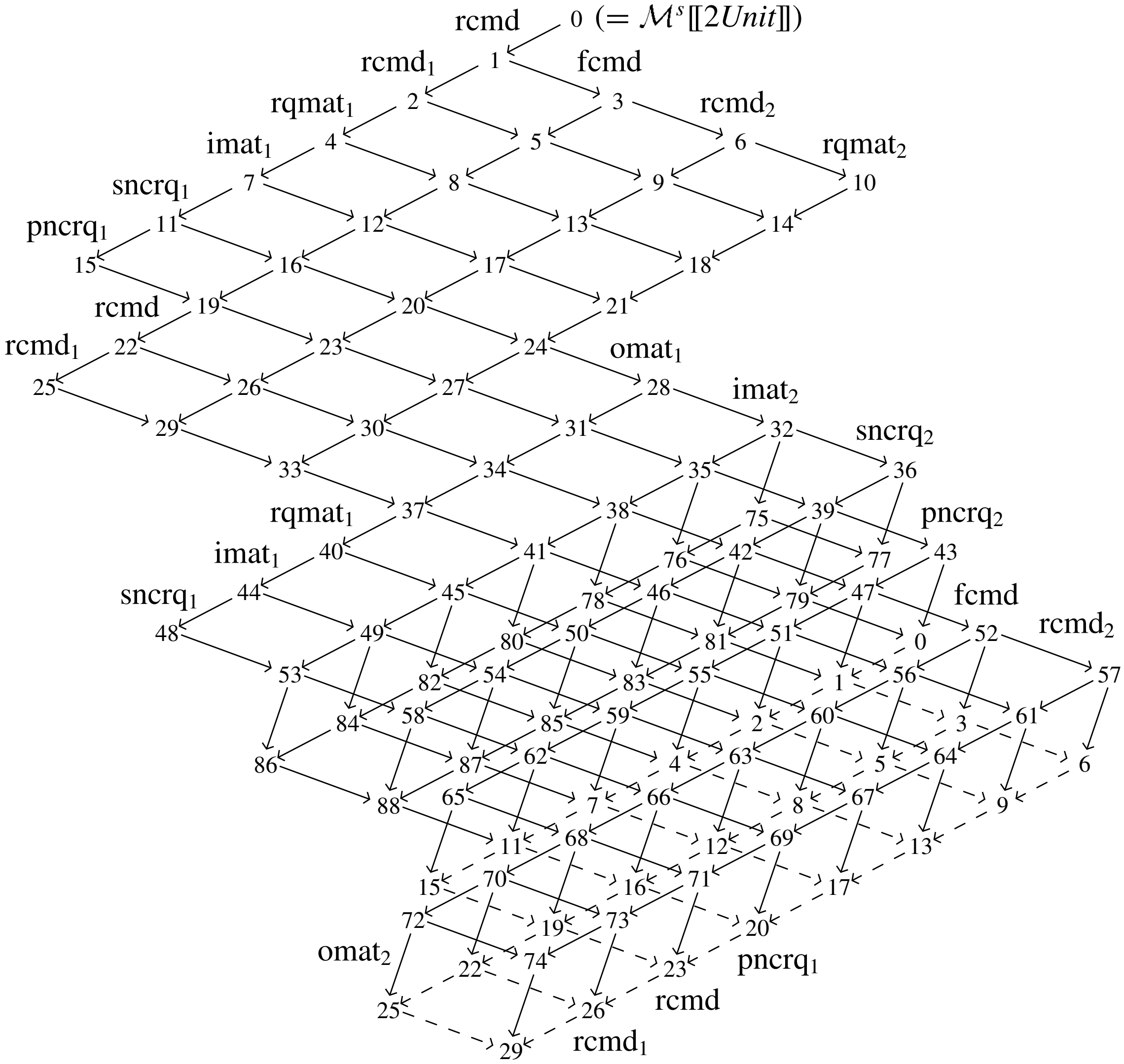


Figure 3.8.41: The system behavior of $2Unit$.

be derived.

$$X_{46} = \text{rqmat}_1 \cdot X_{50} + \text{pncrq}_2 \cdot X_{51} + \text{omat}_2 \cdot X_{83}, \tag{3.8.43}$$

with

$$\begin{aligned} X_{46} &= \lambda_{[ncrq_2]}(\text{pncrq}_2 \cdot C \parallel U_1 \parallel (\text{rcmd}_2 \parallel \text{omat}_2) \cdot U_2), \\ X_{50} &= \lambda_{[ncrq_2]}(\ \text{pncrq}_2 \cdot C \\ &\qquad \parallel \text{imat}_1 \cdot (\text{sncrq}_1 \cdot \text{rcmd}_1 \parallel \text{omat}_1) \cdot U_1 \\ &\qquad \parallel (\text{rcmd}_2 \parallel \text{omat}_2) \cdot U_2 \\ &\qquad), \\ X_{51} &= \lambda_{\mathbf{0}}(C \parallel U_1 \parallel (\text{rcmd}_2 \parallel \text{omat}_2) \cdot U_2), \text{ and} \\ X_{83} &= \lambda_{[ncrq_2]}(\text{pncrq}_2 \cdot C \parallel U_1 \parallel \text{rcmd}_2 \cdot U_2). \end{aligned} \tag{3.8.44}$$

The superscript of the causal state operator is omitted.

Since the algebraic terms characterizing the states of process $2\mathit{Unit}$ all look similar, the complicated part in the calculations is to keep track of all the different states. In addition, one has to recognize states of which the outgoing actions are already calculated. Although this appears to be relatively simple, it would be really helpful to have tool support for these bookkeeping tasks.

As already mentioned, process $\mathcal{M}^s[\![2\mathit{Unit}]\!]$ is recursive. However, it is not iterative. That is, it cannot be expressed in terms of the binary Kleene star. Consequently, the result of the second part of the verification is a set of 89 equations all similar to the one given above for X_{46}.

Verification, Part 3. The third part of the verification consists of hiding the internal consumptions and productions in the system behavior of $2\mathit{Unit}$. Also in this part of the verification, only a few typical steps in the calculations are given. Let I be the set of internal places of $2\mathit{Unit}$, $\{cmd_1, cmd_2, ncrq_1, ncrq_2, trq, tmt\}$. Let T_j, where $0 \leq j \leq 88$, be an abbreviation of $\tau_I(X_j)$. It follows from Result 3.8.38 on Page 128 that T_0 defines the black-box behavior of $2\mathit{Unit}$.

$$T_0 = \tau_I(X_0) = \tau_I(\mathcal{M}^s[\![2\mathit{Unit}]\!]) = \mathcal{M}^b[\![2\mathit{Unit}]\!]. \tag{3.8.45}$$

Consider the list of actions of $2\mathit{Unit}$ given in Table 3.8.40 on Page 129. It is not difficult to see that $\tau_I(\text{rcmd}]) = ([cmd], \mathbf{0})$, $\tau_I(\text{pncrq}_1) = (\mathbf{0}, [ncrq])$, $\tau_I(\text{rqmat}_1) = (\mathbf{0}, [irq])$, $\tau_I(\text{imat}_1) = ([imt], \mathbf{0})$, and $\tau_I(\text{omat}_2) = \text{omat}_2 = ([orq], [omt])$. For all other actions a of $2\mathit{Unit}$, $\tau_I(a) = (\mathbf{0}, \mathbf{0})$, which, as usual, is written τ.

The equations for X_0 through X_{88} only contain the choice and sequential-composition operators. Hence, since the abstraction operator τ_I distributes over these two operators, the actual abstraction of internal consumptions and productions, yielding a set of equations for T_0 through T_{88}, is simple and, therefore, omitted. The tedious part in this step of the verification is the reduction of the set of equations for T_0 through T_{88}. Consider the following subset of the equations that are obtained when hiding the internal consumptions and productions. Actions $([cmd], \mathbf{0})$ and $(\mathbf{0}, [irq])$ are abbreviated *rcmd* and *rqmat*, respectively.

$$
\begin{aligned}
T_0 &= rcmd \cdot T_1, \\
T_1 &= \tau \cdot T_2 + \tau \cdot T_3, \\
T_2 &= rqmat \cdot T_4 + \tau \cdot T_5, \\
T_3 &= \tau \cdot T_5 + \tau \cdot T_6, \\
T_5 &= rqmat \cdot T_8 + \tau \cdot T_9, \\
T_6 &= \tau \cdot T_9 + \tau \cdot T_{10}, \\
T_9 &= rqmat \cdot T_{13} + \tau \cdot T_{14}, \text{ and} \\
T_{10} &= \tau \cdot T_{14}.
\end{aligned} \tag{3.8.46}
$$

The top part of Figure 3.8.47 on the facing page shows the corresponding fragment of the operational semantics for T_0 (see also Figure 3.8.41 on Page 130).

The calculations below show how to reduce the set of equations given in Result 3.8.46. To clarify the reduction, each step is visualized in Figure 3.8.47 on the next page. The idea is to start the reduction in a state in which only a single action can be performed. An example of such a state is T_{10}. State T_{10} corresponds to a "corner" in Figure 3.8.41 on Page 130. Let a be an arbitrary action in $\boldsymbol{A}$, where $\boldsymbol{A}$ is the set of multi-actions defined in Definition 3.3.7 on Page 63.

$$
a \cdot T_{10} \overset{3.8.46}{=} a \cdot \tau \cdot T_{14} \overset{B1}{=} a \cdot T_{14}. \tag{3.8.48}
$$

Informally, Result 3.8.48 says that states T_{10} and T_{14} are equivalent. The initial action a is needed to guarantee that the root condition is satisfied (see Section 2.2.3).

The set of equations in Result 3.8.46 can now be reduced as follows. The equations for T_6 and T_{10} are removed from the set; the equation for T_6, which contains an occurrence of T_{10} in its right-hand side, is replaced by a new equation given below, which is a direct consequence of Result 3.8.48. The equation for T_{10} is no longer needed because T_{10} does not appear anymore on the right-hand side of any of the equations for T_0 through T_{88}.

$$
T_6 = \tau \cdot T_9 + \tau \cdot T_{14}. \tag{3.8.49}
$$

In Figure 3.8.47 on the next page, the consequence of Result 3.8.48 is that the transition $T_6 \xrightarrow{\tau} T_{10}$ changes into a transition $T_6 \xrightarrow{\tau} T_{14}$. As a consequence, state T_{10} is no longer reachable from the initial state T_0.

The following derivation shows that state T_6 is equivalent to state T_9. It uses the new equation derived for T_6.

$$
\begin{aligned}
a \cdot T_6 &\overset{3.8.49}{=} a \cdot (\tau \cdot T_9 + \tau \cdot T_{14}) \overset{3.8.46}{=} a \cdot (\tau \cdot (rqmat \cdot T_{13} + \tau \cdot T_{14}) + \tau \cdot T_{14}) \\
&\overset{B2}{=} a \cdot (rqmat \cdot T_{13} + \tau \cdot T_{14}) \overset{3.8.46}{=} a \cdot T_9.
\end{aligned} \tag{3.8.50}
$$

The set of equations in Result 3.8.46 can now be further reduced. The equation for T_3 is replaced by a new equation given below. The equation for T_6, which was added in the previous reduction step (Result 3.8.49), may be removed from the set again. Note that, when also considering equations not given in Result 3.8.46, the equation for T_{57} must be replaced before the equation for T_6 may be removed (see also Figure 3.8.47 on the facing page).

$$
T_3 = \tau \cdot T_5 + \tau \cdot T_9. \tag{3.8.51}
$$

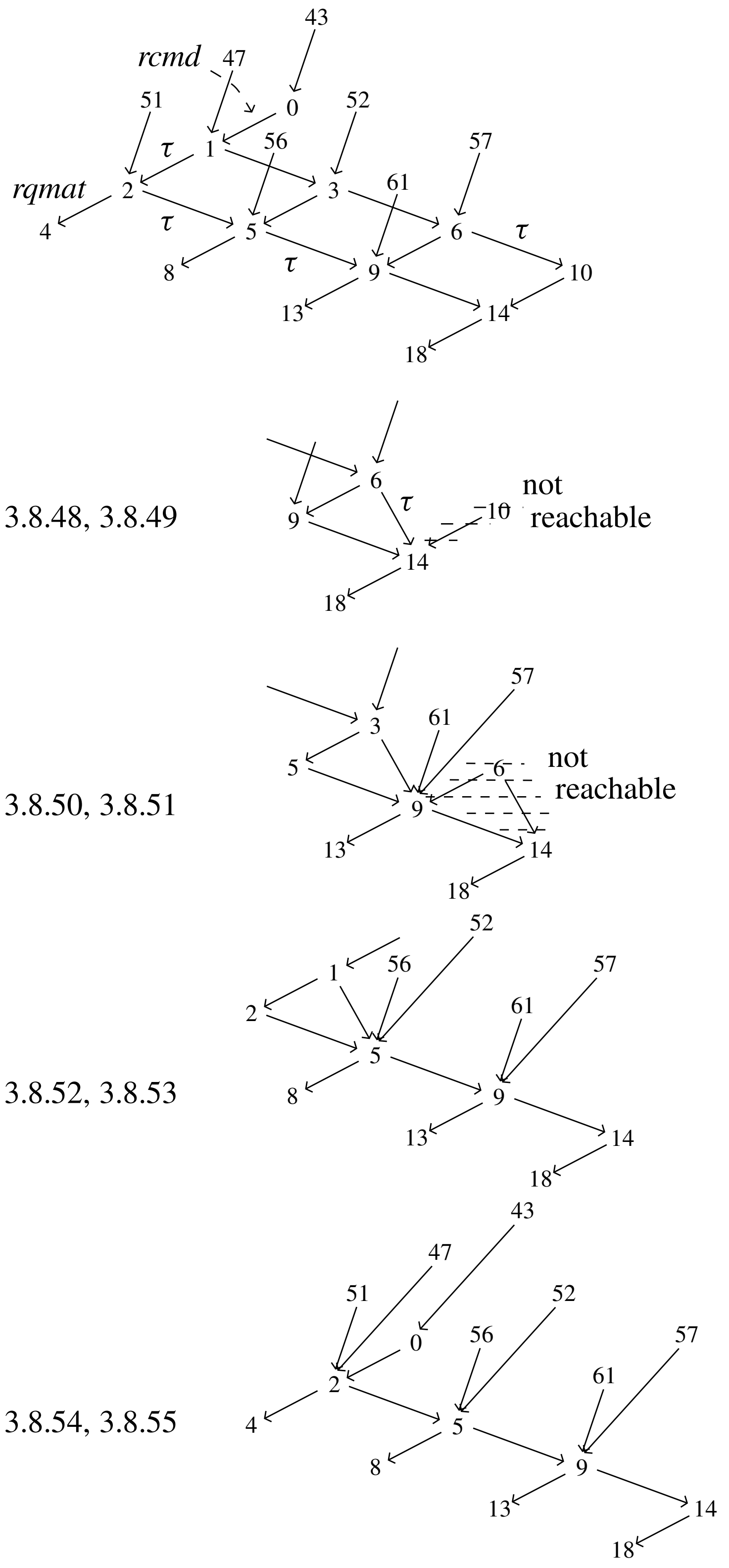


Figure 3.8.47: Hiding internal consumptions and productions of 2*Unit*.

The following step is similar to the previous one.

$$\begin{aligned} a \cdot T_3 &\overset{3.8.51}{=} a \cdot (\tau \cdot T_5 + \tau \cdot T_9) \\ &\overset{3.8.46}{=} a \cdot (\tau \cdot (rqmat \cdot T_8 + \tau \cdot T_9) + \tau \cdot T_9) \overset{B2,3.8.46}{=} a \cdot T_5. \end{aligned} \tag{3.8.52}$$

This result leads to the following new equation, replacing the one for T_1 in Result 3.8.46 on Page 132. The equation for T_3 given in Result 3.8.51 also on Page 132 can be removed.

$$T_1 = \tau \cdot T_2 + \tau \cdot T_5. \tag{3.8.53}$$

The final step is as follows.

$$\begin{aligned} a \cdot T_1 &\overset{3.8.53}{=} a \cdot (\tau \cdot T_2 + \tau \cdot T_5) \\ &\overset{3.8.46}{=} a \cdot (\tau \cdot (rqmat \cdot T_4 + \tau \cdot T_5) + \tau \cdot T_5) \overset{B2,3.8.46}{=} a \cdot T_2. \end{aligned} \tag{3.8.54}$$

This last result leads to a new equation for the initial state T_0. The equation for T_1 added in the previous step is removed. Summarizing all reductions of the equations in Result 3.8.46 on Page 132 yields the following set of equations.

$$\begin{aligned} T_0 &= rcmd \cdot T_2, \\ T_2 &= rqmat \cdot T_4 + \tau \cdot T_5, \\ T_5 &= rqmat \cdot T_8 + \tau \cdot T_9, \text{ and} \\ T_9 &= rqmat \cdot T_{13} + \tau \cdot T_{14}. \end{aligned} \tag{3.8.55}$$

In terms of the operational semantics for T_0 given in Figure 3.8.47 on the preceding page, the reduction amounts to collapsing the entire rectangle between T_1, T_2, T_{10}, and T_{14} into the line from T_2 to T_{14}. Considering Figure 3.8.41 on Page 130 shows that all the silent actions removed in the reduction were the result from one action in the system behavior of $2Unit$, namely rcmd_1.

As the calculations above show, the reduction of the 89 equations for T_0 through T_{88} is not very complicated. Only the bookkeeping is tedious. Essentially, one just collapses rectangles of silent actions as done above until only visible actions remain. The final result of the reduction process is the following set of equations. Together, they characterize the black-box behavior of the modular P/T net $2Unit$ (see also Result 3.8.45 on Page 131).

$$\begin{aligned} \mathcal{M}^b[\![2Unit]\!] = \; T_0 &= ([cmd], \mathbf{0}) \cdot T_{14}, \\ T_{14} &= (\mathbf{0}, [irq]) \cdot T_{18}, \\ T_{18} &= ([imt], \mathbf{0}) \cdot T_{36}, \\ T_{36} &= (\mathbf{0}, [ncrq]) \cdot T_{43} + ([orq], [omt]) \cdot (\mathbf{0}, [ncrq]) \cdot T_0, \\ T_{43} &= ([cmd], \mathbf{0}) \cdot T_{61} + ([orq], [omt]) \cdot T_0, \\ T_{61} &= (\mathbf{0}, [irq]) \cdot T_{64} + ([orq], [omt]) \cdot T_{14}, \\ T_{64} &= ([imt], \mathbf{0}) \cdot T_{69} + ([orq], [omt]) \cdot T_{18}, \\ T_{69} &= (\mathbf{0}, [ncrq]) \cdot T_{71} + ([orq], [omt]) \cdot T_{36}, \text{ and} \\ T_{71} &= ([cmd], \mathbf{0}) \cdot ([orq], [omt]) \cdot T_{61} + ([orq], [omt]) \cdot T_{43}. \end{aligned} \tag{3.8.56}$$

Properties of modular net $2\mathit{Unit}$. The final part of the verification consists of verifying that the black-box behavior of $2\mathit{Unit}$ given in Result 3.8.56 on the preceding page satisfies the behavioral requirements specified in Definitions 3.8.31 and 3.8.32 on Page 123 and 3.8.33 on Page 124. That is, the following three properties have to be verified.

$$\tau_I(\mathcal{M}^b[\![2\mathit{Unit}]\!]) = [\![\mathit{CN}_{\mathrm{spec}}]\!], \tag{3.8.57}$$

where I is the set $\{\mathit{irq}, \mathit{imt}, \mathit{orq}, \mathit{omt}\}$.

$$\tau_I(\mathcal{M}^b[\![2\mathit{Unit}]\!]) = [\![\mathit{IO}_{\mathrm{spec}}]\!], \tag{3.8.58}$$

where I is the set $\{\mathit{cmd}, \mathit{ncrq}, \mathit{irq}, \mathit{orq}\}$.

$$\tau_I(\mathcal{M}^b[\![2\mathit{Unit}]\!]) = [\![\mathit{CM}_{\mathrm{spec}}]\!], \tag{3.8.59}$$

where I is the set $\{\mathit{ncrq}, \mathit{irq}, \mathit{imt}, \mathit{orq}\}$.

The proofs of all three properties are very similar. As an example, the proof of Property 3.8.59 is given in some detail. The proofs of Properties 3.8.57 and 3.8.58 are left to the reader. Essentially, the steps in the proof of Property 3.8.59 are the same as the steps in the computation of the black-box behavior of $2\mathit{Unit}$ from its system behavior in the previous paragraph.

The first step is to hide the appropriate actions in the equations of Result 3.8.56 on the facing page. Let B_j, for $j \in \{0, 14, 18, 36, 43, 61, 64, 69, 71\}$, be an abbreviation of $\tau_I(T_j)$. Consequently,

$$B_0 = \tau_I(T_0) = \tau_I(\mathcal{M}^b[\![2\mathit{Unit}]\!]). \tag{3.8.60}$$

Using the axioms of the abstraction operator, the following set of equations for B_0 is obtained from the equations in Result 3.8.56. Let *rcmd* and *omat* be abbreviations of $([\mathit{cmd}], \mathbf{0})$ and $(\mathbf{0}, [\mathit{omt}])$, respectively.

$$\begin{aligned}
B_0 &= \mathit{rcmd} \cdot B_{14}, \\
B_{14} &= \tau \cdot B_{18}, \\
B_{18} &= \tau \cdot B_{36}, \\
B_{36} &= \tau \cdot B_{43} + \mathit{omat} \cdot \tau \cdot B_0, \\
B_{43} &= \mathit{rcmd} \cdot B_{61} + \mathit{omat} \cdot B_0, \\
B_{61} &= \tau \cdot B_{64} + \mathit{omat} \cdot B_{14}, \\
B_{64} &= \tau \cdot B_{69} + \mathit{omat} \cdot B_{18}, \\
B_{69} &= \tau \cdot B_{71} + \mathit{omat} \cdot B_{36}, \text{ and} \\
B_{71} &= \mathit{rcmd} \cdot \mathit{omat} \cdot B_{61} + \mathit{omat} \cdot B_{43}.
\end{aligned} \tag{3.8.61}$$

The second step is to remove silent actions and reduce the set of equations. By means of Axiom $B1$ and repeated substitution, the following reduction can be obtained. For the sake of convenience, equations that have changed in comparison to Result 3.8.61 are marked by an asterisk.

$$\begin{aligned}
* \; B_0 &= \mathit{rcmd} \cdot B_{36}, \\
* \; B_{36} &= \tau \cdot B_{43} + \mathit{omat} \cdot B_0, \\
B_{43} &= \mathit{rcmd} \cdot B_{61} + \mathit{omat} \cdot B_0, \\
* \; B_{61} &= \tau \cdot B_{64} + \mathit{omat} \cdot B_{36}, \\
* \; B_{64} &= \tau \cdot B_{69} + \mathit{omat} \cdot B_{36},
\end{aligned} \tag{3.8.62}$$

$$\begin{aligned} B_{69} &= \tau \cdot B_{71} + omat \cdot B_{36}, \text{ and} \\ B_{71} &= rcmd \cdot omat \cdot B_{61} + omat \cdot B_{43}. \end{aligned} \tag{3.8.62) ctd.$$

In the third step, Axiom $B2$ is used to obtain a further reduction. Let a be an arbitrary action in $\boldsymbol{A}$. By means of derivations similar to the ones in the previous paragraph, the following results can be proved.

$$\begin{aligned} & a \cdot B_{36} = a \cdot B_{43} \text{ and} \\ & a \cdot B_{61} = a \cdot B_{64} = a \cdot B_{69}. \end{aligned} \tag{3.8.63}$$

Repeated substitution yields the following reduction of the equations in Result 3.8.62.

$$\begin{aligned} * \; B_0 &= rcmd \cdot B_{43}, \\ * \; B_{43} &= rcmd \cdot B_{69} + omat \cdot B_0, \\ * \; B_{69} &= \tau \cdot B_{71} + omat \cdot B_{43}, \text{ and} \\ * \; B_{71} &= rcmd \cdot omat \cdot B_{69} + omat \cdot B_{43}. \end{aligned} \tag{3.8.64}$$

Applying Axiom $B2$ again yields the following equation.

$$a \cdot B_{69} = a \cdot B_{71}, \tag{3.8.65}$$

This last result leads to the final reduction.

$$\begin{aligned} B_0 &= rcmd \cdot B_{43}, \\ * \; B_{43} &= rcmd \cdot B_{71} + omat \cdot B_0, \text{ and} \\ * \; B_{71} &= rcmd \cdot omat \cdot B_{71} + omat \cdot B_{43}. \end{aligned} \tag{3.8.66}$$

Definitions 3.7.1 (Interpretation $[\![\_]\!]$) on Page 90 and 3.8.33 (Specification CM_{spec}) on Page 124 plus substitution yield the following algebraic interpretation of CM_{spec}.

$$\begin{aligned} [\![CM_{\text{spec}}]\!] &= rcmd \cdot [\![CM_1]\!], \\ [\![CM_1]\!] &= rcmd \cdot [\![CM_2]\!] + omat \cdot [\![CM_{\text{spec}}]\!], \text{ and} \\ [\![CM_2]\!] &= rcmd \cdot omat \cdot [\![CM_2]\!] + omat \cdot [\![CM_1]\!]. \end{aligned} \tag{3.8.67}$$

The result of the verifications so far are two sets of identical recursive equations (Results 3.8.66 and 3.8.67). Assuming that this leads to equation

$$B0 = [\![CM_{\text{spec}}]\!], \tag{3.8.68}$$

Property 3.8.59 on the previous page, which is the one to be proven, follows immediately from Result 3.8.60 also on the previous page.

However, one has to be careful with conclusions such as the one in Property 3.8.68. So far, recursive equations have always had a solution in terms of the binary Kleene star, meaning that they describe some sort of iteration. The recursion principle RSP^* is used to get such solutions of recursive equations in terms of the Kleene star. However, the two sets of equations in Results 3.8.66 and 3.8.67 do not describe simple iterations. They do not have solutions in terms of the binary Kleene star. In the current framework, it is not formalized what solutions the above two sets of equations have. They might, for example, have two different solutions, in which case it is *not* allowed to draw the conclusion in Property 3.8.68. In order to prove anything about the solutions of the recursive equations in Results 3.8.66 and 3.8.67, it is necessary to extend the theory developed so far with general recursion as treated in (BAETEN AND VERHOEF, 1995; BAETEN AND WEIJLAND, 1990). However, it goes beyond the scope of this thesis to go into details. To prove Property 3.8.68, the *Recursive Specification Principle* (*RSP*) is needed. Principle *RSP* has already been mentioned in Section 2.4.5. It is a generalization of RSP^* stating that (guarded) recursive equations have a unique solution. Assuming *RSP*, Property 3.8.68 on the preceding page follows immediately, completing the proof of Property 3.8.59 on Page 135.

The proofs of Properties 3.8.57 and 3.8.58 on Page 135 are very similar to the proof of Property 3.8.59, except that RSP^* is used instead of *RSP*.

Concluding remarks. In this subsection, a generic control component has been developed which can be used to concatenate basic units as developed in Section 3.8.2 into series of arbitrary length, called complex units. Instead of verifying the complete behavior of such complex units against a specification, three behavioral requirements for complex units have been derived from the behavior of basic units. It has been shown that the complex unit consisting of two basic units in series, modeled by the modular P/T net $2Unit$ of Figure 3.8.30 on Page 122, satisfies these behavioral requirements. Simulations for a complex unit consisting of three basic units in series have not revealed any results contradicting the behavioral requirements for such a complex unit. The verification of the requirements for complex units of arbitrary length remains future work. Such a verification is only possible with proper tool support and if the theory developed in this chapter is extended with data.

As mentioned in the introduction, the production unit developed in this section is a simplified version of the production unit in the real case. In this section, it has been assumed that no processing errors can occur. In reality this is not the case. When processing errors are taken into account, the number of states of the net model of the production unit becomes so large that verification by hand is almost impossible. This confirms the conclusion that tools are needed to support verifications of real-world concurrent systems.

In the real case, it was also required that units could be placed in parallel. To satisfy this requirement, a second control component is needed. Assuming that no processing errors can occur, the complexity of the design process for this component is the same as the complexity of the design process for the control component for units in series. Verifying complex units combining both units in series and in parallel is not possible without tool support. When processing errors are taken into account, the complexity of the verifications increases even further.

3.9 Conclusions

Concluding remarks. In this chapter, a method for the compositional design of concurrent systems and an accompanying formalism combining Petri nets and process algebra have been developed. Section 3.2 has briefly introduced the design method. In Sections 3.3 through 3.6 the theoretical foundations have been given. To validate the method and the underlying theory, two case studies were performed. The results have been given in Sections 3.7 and 3.8.

The design method of Section 3.2 focuses on the correct design of the communication behavior of concurrent systems. For this purpose, it combines four activities: behavioral specification, Petri-net modeling, simulation, and algebraic verification. The goal is to structure the design process and improve the understanding of the concurrent system under development. The method is not meant for data-oriented applications with only little communication between components of the concurrent system.

The basis of the theory underlying the design method is the notion of modular P/T nets introduced in Section 3.5. A modular P/T net is a simple form of hierarchical Petri net with only two hierarchical levels. The building blocks of modular P/T nets are the so-called P/T-net components, as defined in Section 3.3. In Section 3.4, two algebraic semantics for P/T-net components have been given, describing their glass-box and black-box behavior, respectively. It has been shown that these semantics are consistent with the usual (interleaving) semantics for P/T nets. In Section 3.6, three algebraic semantics for modular P/T nets have been given. These semantics describe the glass-box behavior, system behavior, and black-box behavior of modular P/T nets, respectively. They are defined in terms of the algebraic semantics for P/T-net components. Hence, they are truly compositional. As for P/T-net components, it is shown that the algebraic semantics for modular P/T nets conform to the standard semantics for P/T nets.

As already mentioned, the method of this chapter has been applied to two case studies. The first case study is the design and verification of the Alternating-Bit Protocol. Since the ABP is a simple, well-understood protocol, it serves well to illustrate the new method and the underlying formalism. The case study shows that the design method satisfies its goals. It provides convenient ways to structure the design process. In addition, the combination of a Petri-net model and process-algebraic verifications provides a deep insight in all details of the protocol. However, the verification of the ABP shows also that the algebraic calculations are tedious. The question arises whether the formalism can be applied to the development of systems more complex than the ABP. Note that the design method of Section 3.2 is more or less independent of the specific formalism chosen for the verifications. The formalism can be adapted without having to change the design method.

The second case study describes the design of a production unit. Contrary to the ABP, the design problem is not trivial. The case study confirms that the design method is useful in structuring the design process, particularly, if it is applied in a flexible way. The design method is a means to arrive at a correct system, not a goal in itself. Unfortunately, the case study also confirms that the verifications are tedious. The axiomatization of the causal state operator forces the designer to calculate each single action that the system under development can perform. The verifications of the production unit show that, for complex concurrent systems, this is not possible without proper tool support. In addition, the case study shows that it would be useful to integrate data in the theory in the style of the process algebra μCRL (GROOTE AND PONSE, 1994). Such an extension allows parameterized and inductive verifications, making it possible to verify properties of many variations of a system in one verification.

Future work. The experience so far with the method of this chapter shows that at least two extensions are needed before it will become useful in the design of real-world concurrent systems. The first one is tool support. The second one is the integration of data in the underlying theory. Only if both extensions are realized, the method may be able to compete with other verification methods for concurrent systems, such as model checking techniques. Particularly the possibility of parameterized and inductive verifications would be an advantage over model checking. A preliminary study into the possibilities for tool support for equational reasoning by means of the generic proof system PVS (OWRE ET AL., 1995) can be found in (BASTEN AND HOOMAN, 1998). This study shows that, in principle, it is feasible to develop automated proof support for large equational verifications. However, at the time of doing the research described in (BASTEN AND HOOMAN, 1998), a bug in the type system of PVS prevented extensive experiments with the theory described in this chapter. The extension of the theory with data remains an open problem. It is not yet clear how to axiomatize the interaction between the causal state operator and the sum operator of μCRL. The sum operator is crucial in reasoning about infinite data types, such as the natural numbers.

Other useful extensions of the theory are possible. However, they are not as crucial as the ones mentioned above. A standard way to alleviate the state explosion in the verification of systems with concurrent behavior is the use of priorities. Such an extension would not only simplify verifications, but it would also increase the modeling power of the formalism.

Another extension could be the incorporation of the weak-sequential-composition operator of (RENSINK AND WEHRHEIM, 1994) in the theory. Weak sequential composition is a process-algebraic operator lying in between the sequential-composition operator and the merge operator. It is parameterized with a dependency relation on actions. If two processes are composed by means of a weak sequential composition, independent actions may occur concurrently, whereas dependent actions must happen sequentially. Modular P/T nets that are composed via places often exhibit such partially concurrent and partially sequential behavior. A good example is the modular P/T net $2\mathit{Unit}$ of Figure 3.8.30 on Page 122. Its system behavior depicted in Figure 3.8.41 on Page 130 is typical for modular P/T nets. The experienced reader easily recognizes several "almost-concurrent" parts in the behavior, as well as a few "almost-sequential" parts. However, note that the weak-sequential-composition operator is useful only to concisely represent the behavior of modular P/T nets. It cannot simplify the actual verifications.

Bibliographical remarks. The research described in this chapter is motivated by the development of the tool ExSpect (ASPT, 1994), which has already been mentioned several times earlier. ExSpect supports the construction, simulation, and analysis of colored-Petri-net models of complex concurrent systems. ExSpect was first described in (VAN HEE, SOMERS, AND VOORHOEVE, 1989). Its theoretical foundation is described in detail in (VAN HEE, 1994). A good example of the use of ExSpect in practical applications can be found in (BASTEN, BOL, AND VOORHOEVE, 1995). During the development of ExSpect, it became clear that a useful extension of the tool would be the possibility to verify the interface behavior of system components in a compositional way. The theoretical basis for such an extension was first investigated in (RAMBAGS, 1994). The approach taken in (RAMBAGS, 1994) is to find algorithms for deciding in a fully automated way whether a Petri net satisfies a certain prescribed behavior. However, this algorithmic approach proves to have its limitations, because the general problem whether a (colored) Petri net satisfies a certain behavior is undecidable. For this reason, it was decided to pursue a different approach in which the verification of a Petri-net model against its behavioral specification is done in an interactive way. For reasons explained earlier, process algebra was chosen to support such an approach.

The literature on concurrency theory contains many attempts to integrate Petri nets and process algebra. One line of research is concerned with the translation of process-algebraic terms into Petri nets. The aim of this approach is to provide the process algebra with a true-concurrency semantics. Examples of this line of research are (BEST, DEVILLERS, AND HALL, 1992), (DEGANO, DE NICOLA, AND MONTANARI, 1988), (VAN GLABBEEK AND VAANDRAGER, 1987), (GOLTZ, 1988), (MONTANARI AND YANKELEVICH, 1994), (OLDEROG, 1987), and (TAUBNER, 1989).

In this chapter, the converse approach is pursued. A translation from Petri nets into algebraic terms is given. Other examples of this approach are (BAETEN AND BERGSTRA, 1993), (BOUDOL, ROUCAIROL, AND DE SIMONE, 1985), and (DIETZ AND SCHREIBER, 1994). The approach followed in each of these three papers is discussed briefly.

In (DIETZ AND SCHREIBER, 1994), an algebraic semantics for P/T nets is given which reflects the parallelism in their dynamic behavior. The parallel components in the algebraic representation of a net do not necessarily correspond to its structural components, as is the case in the semantics given in Section 3.4.

A structural relationship between a Petri net and its algebraic semantics also exists in (BOUDOL, ROUCAIROL, AND DE SIMONE, 1985). In that paper, an algebraic term is given for each place and each transition of a P/T net. The complete behavior of the net is the parallel composition of all these terms. The communication between these terms corresponds to the flow of tokens. The semantics of this chapter are also based on the flow of tokens. However, to simulate the token flow, the causal state operator is used instead of a communication mechanism. Another difference is that, in this chapter, equational reasoning is emphasized, whereas in (BOUDOL, ROUCAIROL, AND DE SIMONE, 1985) reasoning is mainly model-based.

The algebraic semantics of this chapter are most closely related to the semantics given in (BAETEN AND BERGSTRA, 1993). In that paper, the algebraic semantics of P/T nets is also defined in an ACP-like process algebra, emphasizing an equational style of reasoning. As already mentioned in Section 3.4, (BAETEN AND BERGSTRA, 1993) is also the first paper to introduce a variant of the causal state operator. In the semantics of (BAETEN AND BERGSTRA, 1993), atomic actions in the algebra correspond to transitions in P/T nets. So-called input and output causes are added to these actions, corresponding to input and output places. Similar to the semantics in this chapter, the behavior of a net is the parallel composition of the algebraic expressions corresponding to its transitions, restricted by means of the causal state operator.

Contributions. The first contribution of this chapter is that a number of algebraic semantics is given for *modular* P/T nets. A simple, but crucial aspect of these semantics is that actions in the process algebra are structured (see Definition 3.3.7 (Multi-actions) on Page 63). Each action is a pair of bags. Each element of these bags corresponds to the consumption or the production of a single token. Since the interface of modular nets consists of places, it is not clear how to generalize a semantics for P/T nets without hierarchy to modular P/T nets when actions in the algebra correspond to transitions. When actions are structured as in this chapter, the generalization is straightforward, as is shown in Section 3.5.

The second contribution of this chapter is that a first attempt is made to develop a design method based on a formalism integrating Petri nets and process algebra. Most other results on the integration of Petri nets and process algebra appearing in the literature are mainly theoretical. An exception is the Petri Box Calculus of (BEST, DEVILLERS, AND HALL, 1992) which has inspired a whole line of research, including the development of tools and the basics of a design method (GRAHLMANN AND BEST, 1996). However, there are many differences with the approach taken in this chapter. In (BEST, DEVILLERS, AND HALL, 1992; GRAHLMANN AND BEST, 1996), the starting point is a programming language based on process-algebraic principles. Petri nets are used to provide this language with a semantics. The analysis techniques of Petri nets are used to verify properties of the system under development. In this chapter, the converse approach is taken. Petri nets serve as the modeling formalism. Process algebra is used for the purpose of verification. Both approaches are interesting and deserve further study.

4

Inheritance of Dynamic Behavior

4.1 Introduction

To date, a popular approach to the modular design of complex systems is the so-called *object-oriented* approach. The modularity construct in any object-oriented design method is the *class* construct. A *class* describes a set of *objects* with a common structure and behavior. An *object* is an *instance* of a class. Classes may, for example, describe persons, cars, or production units. Objects of such classes are typically person X, car Y, or production unit Z. Each class and, hence, each object has a set of *attributes*. Attributes describe properties of objects. The *value* of these attributes determines the *state* of an object. In addition, each class has a set of *methods*. A *method* is an operation on an object. Methods may, for example, be used to read the value of an attribute, or to change the state of an object. Finally, a class contains a definition of the dynamic behavior of objects. That is, it specifies the order in which the methods of an object may be executed. Such a specification is called the *life cycle* of an object.

Three object-oriented methods that are in common use are OMT (RUMBAUGH ET AL., 1991), OOD (BOOCH, 1994), and UML (FOWLER AND SCOTT, 1997). The given references are a good starting point for the reader interested in an introduction to object-oriented design including a detailed explanation of all the abovementioned concepts.

One of the main goals of object-oriented design is the *reuse* of system components. A key concept to achieve this goal is the concept of *inheritance*. The inheritance mechanism allows the designer to specify a class, the *subclass*, that inherits features of some other class, its *superclass*. Thus, it is possible to specify that the subclass has the same features as the superclass, but that in addition it may have some other features.

The concept of inheritance is usually well defined for the *static structure* of a class consisting of the set of methods and the attributes. However, as mentioned, a class contains also a definition of the dynamic behavior of an object, the object life cycle. Techniques as OMT, OOD, and UML use state-transition diagrams for specifying object life cycles.

Such a diagram shows the state space of an object and the method calls that cause a transition from one state to another. Looking at the definition of inheritance in, for example, OOD and its informal explanation, a subclass that inherits features of some other class extends the *static structure* as well as the *dynamic behavior* of its parent class. However, in the further treatment of inheritance, OOD only defines inheritance of static features. It does not specify the meaning of inheritance of dynamic behavior. It is implicitly assumed that the behavior of the objects of a subclass is an *extension* of the behavior of the objects of its superclass. OOD does not further elaborate on the precise meaning of "extension."

Consider two classes $Unit_1$ and $Unit_2$ modeling production units. Both classes have the same methods, namely $pmat_1$ and $pmat_2$, modeling two processing operations on input material. Objects of class $Unit_1$ first execute $pmat_1$ and then $pmat_2$. Objects of class $Unit_2$ perform either $pmat_1$ or $pmat_2$ but not both. Should one of the classes $Unit_1$ or $Unit_2$ be a subclass of the other one? Although the two classes have the same set of methods, their behavior is clearly different. Hence, the answer to the above question should be negative.

Therefore, this chapter studies several formalizations of what it means for an object life cycle to extend the behavior prescribed by another object life cycle. Combining the usual definition of inheritance of methods and attributes with a definition of inheritance of dynamic behavior yields a complete formal definition of inheritance, thus, stimulating the reuse of life-cycle specifications during the design process. However, it is beyond the scope of this thesis to develop a complete object-oriented method including a notion of inheritance of life cycles. Instead, this chapter focuses on the fundamentals of inheritance of dynamic behavior. The integration of the results in a full fledged object-oriented design method is left for future work.

As mentioned, OMT, OOD, and UML use state-transition diagrams to specify object life cycles. Although the graphical nature and the explicit representation of states are essential to the success and usefulness of the abovementioned methods, particularly the latter impedes a clear understanding of life-cycle inheritance. For studying inheritance of dynamic behavior, the most important aspects of a life cycle are the *state changes* and not the states themselves. Therefore, the problem of inheritance of dynamic behavior is first studied in a process-algebraic setting. In general, a process algebra does not have an explicit representation of process states. In Section 4.2, it is argued that inheritance of life cycles corresponds to *hiding* and *blocking* method calls. In process-algebraic terms, the former corresponds to *abstraction* and the latter to *encapsulation*. Encapsulation and abstraction have been introduced in Chapter 2 and are well understood in the context of process algebra. Note that the terms "abstraction" and "encapsulation" in process algebra have a different meaning than the same terms in object-oriented design. In this chapter, they always refer to the process-algebraic concepts. Section 4.3 translates the results of Section 4.2 to Petri nets. Petri nets have a solid theoretical basis and, due to their explicit representation of process states and their graphical nature, they are close to the state-transition diagrams used in practical object-oriented methods as OMT, OOD, and UML. The translation is illustrative for translations of the fundamentals developed in Section 4.2 to other graphical, state-based formalisms. To validate the approach to inheritance of dynamic behavior chosen in this chapter, Section 4.4 discusses a case study. It describes the use of inheritance in the design of a groupware editor. Finally, Section 4.5 discusses some conclusions and open problems.

4.2 A Process-Algebraic Approach

The goal of this section is to characterize the essential properties of inheritance of dynamic behavior. For this purpose, only the object life cycle of a class is considered. All other aspects of a class are omitted. As explained, it must be formalized when an object life cycle of some class extends the object life cycle of another class. In the next subsection, four possible formalizations are given, yielding four different inheritance relations on object life cycles. Each formalization characterizes a different type of extension. In Section 4.2.2, equational laws preserving each of the inheritance relations are presented. These so-called *axioms of inheritance* are useful for several purposes. First, they illustrate the characteristic properties of each of the inheritance relations. Second, given two object life cycles, they can be used to verify in an algebraic framework whether one life cycle is a subclass of the other life cycle. Finally, they can be used as *transformation rules* to *construct* subclasses of life cycles. Thus, they stimulate the reuse of life-cycle specifications during the design process. In Section 4.3, the axioms of inheritance are used as a source of inspiration for transformation rules on object life cycles in a framework of Petri nets.

4.2.1 Object life cycles and inheritance relations

The first step in formalizing the concept of inheritance of dynamic behavior is to define the notion of an object life cycle. As explained, the life cycle of an object describes the order in which the methods of the object may be executed. Since the implementation details of methods are not relevant in this chapter, a method is simply represented by an identifier. Usually, a method belonging to some object interacts with the environment of the object. Such a method is called an *external* method. The order in which external methods may be executed determines the *external behavior* of an object. Sometimes, a method can only be executed by the object itself and has only internal effects. Such a method does not contribute to the external behavior of an object. Internal methods are denoted by the special identifier τ. The set of external methods is denoted E. The set of all methods is denoted M. That is, M equals $E \cup \{\tau\}$.

It should not come as a surprise that, in this section, an object life cycle is specified by a process-algebraic term. The question is what process algebra is best suited to specify object life cycles. The set of actions of the process algebra corresponds to the set of methods M. Since M includes the internal method τ that behaves as a silent action, the process algebra must include silent actions. Operators that are useful in the specification of object life cycles are choice, sequential composition, and parallel composition. From a modeling viewpoint, iteration in the form of the binary Kleene star would also be useful. However, the main goal of this section is to study the essential properties of inheritance of dynamic behavior. As explained in Chapter 2 on Page 52, iteration complicates reasoning in a process-algebraic framework. Therefore, in this section, iteration is not allowed in specifications of object life cycles. Iterative behavior in life cycles is studied in the next section, where object life cycles are defined in a Petri-net framework. In conclusion, a process algebra well suited for specifying and analyzing object life cycles is the theory PA^{τ}, as introduced in Section 2.4.4, instantiated with the set of methods M.

Definition 4.2.1. (Object life cycle) An object life cycle is a closed PA^{τ} term.

Example 4.2.2. The life cycle of a simple production unit U can be described by the algebraic term $\mathit{rcmd} \cdot \mathit{pmat} \cdot \mathit{omat}$. The production unit first receives a command. After processing some piece of raw material, it delivers processed output material.

Having defined the notion of an object life cycle, the next step is to formalize when one class inherits the life cycle of another class. For this purpose, the question must be answered when one life cycle extends the behavior of another life cycle. It is important to answer this question from the viewpoint of the environment of an object. Only the external behavior of an object is relevant in defining inheritance relations.

Recall that the set of closed terms of a process algebra X is denoted $\mathcal{C}(X)$. Assume that p and q in $\mathcal{C}(\mathrm{PA}^{\tau})$ are two object life cycles such that p contains all methods of q and, possibly, several more. In other words, p inherits all methods of q. The following appears to be a meaningful answer to the above question.

> *If it is not possible to distinguish the external behavior of p and q when only methods of p that are also present in q are executed, then p is a subclass of q.*

Intuitively, this definition conforms to *blocking* or *encapsulating* methods new in p. Life cycle p is said to inherit the *protocol* of q.

As explained in the introduction to this chapter, blocking and also hiding method calls play an important role in reasoning about inheritance of dynamic behavior. Therefore, the process algebra used to formalize inheritance of behavior is the theory $\mathrm{PA}^{\tau}_{\delta}+\mathrm{RN}$ as defined in Section 2.4.4. Process algebra $\mathrm{PA}^{\tau}_{\delta}+\mathrm{RN}$ extends the theory PA^{τ}, used to define object life cycles, with encapsulation and abstraction. These two operators are exactly the operators needed to reason about inheritance. Theory $\mathrm{PA}^{\tau}_{\delta}+\mathrm{RN}$ is instantiated with the set of method identifiers M.

Example 4.2.3. Consider again Example 4.2.2. The operator who is responsible for issuing commands to the production unit sometimes makes a mistake and sends the wrong command. Therefore, production unit U of Example 4.2.2 is extended with an error-handling facility. A new method *error* is added that is executed when the unit does not understand the command it receives. The behavior of the new unit U_1 is described as follows: $\mathit{rcmd} \cdot (\mathit{pmat} \cdot \mathit{omat} + \mathit{error})$. Since the addition of an error-handling facility should not influence the correct behavior of the unit, the new unit should be a subclass of U. It is not difficult to see that the behavior of the two production units is identical when the new method *error* is not executed. Hence, according to the above definition, production unit U_1 is indeed a subclass of unit U of Example 4.2.2.

To show that the above definition conforms to encapsulation, let $H \subseteq E$ be the singleton $\{error\}$. It follows easily from the axioms for encapsulation of the process algebra $\mathrm{PA}^{\tau}_{\delta}$+RN and Axiom $A6$ that $\partial_H(rcmd \cdot (pmat \cdot omat + error)) = rcmd \cdot pmat \cdot omat$. That is, $\partial_H(U_1) = U$.

The definition above and the accompanying example introduce one possible definition of inheritance of dynamic behavior. However, other definitions are possible. A second one that appears to be meaningful is the following. Let p and q in $\mathcal{C}(\mathrm{PA}^{\tau})$ be two object life cycles such that p contains all methods of q.

> *If it is not possible to distinguish the external behavior of p and q when arbitrary methods of p are executed, but when only the effects of methods that are also present in q are considered, then p is a subclass of q.*

This second definition of inheritance of behavior conforms to *hiding* the effect of methods new in p. Life cycle p inherits the *projection* of the life cycle of p onto the methods of q.

Example 4.2.4. Consider a production unit U_2 that between the receipt of a command and the main processing step performs a preprocessing step on the raw input material. Unit U_2 is specified as follows: $rcmd \cdot ppmat \cdot pmat \cdot omat$. Clearly, in some sense, the behavior of U_2 extends the behavior of production unit U of Example 4.2.2 on the facing page. Therefore, U_2 should be a subclass of U. It is not difficult to see that the behaviors of units U and U_2 are identical when the new method *ppmat* of U_2 is executed, but its effect is ignored. Hence, according to the second definition of inheritance, U_2 is indeed a subclass of U. It is also not difficult to see that the second form of inheritance conforms to hiding method calls. Let I be the singleton $\{ppmat\}$. It follows from the abstraction axioms and Axiom $B1$ that $\tau_I(U_2) = U$.

The subtle difference between the two forms of inheritance introduced above is that in the second definition methods new in life cycle p are executed without taking into account their effect, whereas in the first definition they are not executed at all. The examples in the remainder of this section further illustrate this difference.

There are several more meaningful definitions of inheritance relations for dynamic behavior than the two definitions given so far. It is, for example, possible to combine the two definitions given above. One might argue that for methods new in the subclass the requirements of both definitions must hold at the same time, or one might argue that for some methods the first requirement must hold, whereas for some other methods the other requirement holds. The two basic definitions of inheritance given above and the two combinations yield four possible inheritance relations. Although probably other meaningful relations can be found, this chapter focuses on these four relations. An attempt is made to show that they capture the essentials of inheritance of dynamic behavior.

The formal definitions of the four inheritance relations given below are slightly more general than the informal definitions: A life cycle is a subclass of another life cycle if and only if there exists *some* set of methods such that encapsulating or hiding these methods in the first life cycle yields the other life cycle. Not requiring that the methods being encapsulated or hidden must be *exactly* the methods appearing in the first life cycle and not in the second one can sometimes be convenient, as some of the examples in the remainder show.

Before going to the formal definition of the four inheritance relations, a note on notation is in order. Recall from Sections 2.4.3 and 2.4.4, that the encapsulation operator and the abstraction operator both belong to the class of algebraic renaming operators. Many formulas in this section contain multiple occurrences of these two renaming operators. To avoid large numbers of brackets in formulas, the notation for function composition is overloaded to renaming operators. Let x be a PA_δ^τ+RN term and let $\boldsymbol{f}, \boldsymbol{g} \in \boldsymbol{F}$ be renaming functions as introduced in Section 2.4.3. The notation $\rho_{\boldsymbol{f}} \circ \rho_{\boldsymbol{g}}(x)$ is an abbreviation for the algebraic term $\rho_{\boldsymbol{f}}(\rho_{\boldsymbol{g}}(x))$.

Definition 4.2.5. (Inheritance relations)

i) *Protocol inheritance*:
For any object life cycles $p, q \in \mathcal{C}(\mathrm{PA}^\tau)$, life cycle p is said to be a subclass of q under *protocol inheritance*, denoted $p \leq_{pt} q$, if and only if there exists an $H \subseteq E$ such that $\mathrm{PA}_\delta^\tau+\mathrm{RN} \vdash \partial_H(p) = q$.

ii) *Projection inheritance*:
For any object life cycles $p, q \in \mathcal{C}(\mathrm{PA}^\tau)$, life cycle p is a subclass of q under *projection inheritance*, denoted $p \leq_{pj} q$, if and only if there exists an $I \subseteq E$ such that $\mathrm{PA}_\delta^\tau+\mathrm{RN} \vdash \tau_I(p) = q$.

iii) *Protocol/projection inheritance*:
For any object life cycles $p, q \in \mathcal{C}(\mathrm{PA}^\tau)$, life cycle p is a subclass of q under *protocol/projection inheritance*, denoted $p \leq_{pp} q$, if and only if there exists an $H \subseteq E$ such that $\mathrm{PA}_\delta^\tau+\mathrm{RN} \vdash \partial_H(p) = q$ *and* there exists an $I \subseteq E$ such that $\mathrm{PA}_\delta^\tau+\mathrm{RN} \vdash \tau_I(p) = q$.

iv) *Life-cycle inheritance*:
For any object life cycles $p, q \in \mathcal{C}(\mathrm{PA}^\tau)$, life cycle p is a subclass of q under *life-cycle inheritance*, denoted $p \leq_{lc} q$, if and only if there exist *disjoint* subsets $H, I \subseteq E$ such that $\mathrm{PA}_\delta^\tau+\mathrm{RN} \vdash \tau_I \circ \partial_H(p) = q$.

The above definitions are formulated in terms of equality of closed PA_δ^τ+RN terms. The completeness of PA_δ^τ+RN for rooted branching bisimilarity (Theorem 2.4.45 on Page 51) implies that this formulation is equivalent to a formulation in terms of rooted branching bisimilarity. Without completeness, the above definitions had been too restrictive. It would have been possible that an object life cycle p after encapsulation and/or abstraction of the appropriate methods and life cycle q would be rooted branching bisimilar, but that this equality would not be derivable from the axioms of PA_δ^τ+RN. This would be undesirable, because, according to the above definitions, p would not be a subclass of q.

The requirement that H and I must be disjoint in the definition of life-cycle inheritance means that methods are either consistently encapsulated or consistently hidden. It implies that the order of encapsulation and abstraction can be changed without actually changing the definition (see also Lemma 4.2.14 on Page 152). It is not clear whether it is meaningful to treat different calls of one method in a different way. In the current definition, it is not possible to hide some calls of a method in some part of an object life cycle, whereas some other calls of the same method in another part of the life cycle are encapsulated or left untouched. In the case study of Section 4.4, it proves not to be necessary to treat different calls of the same method differently when determining inheritance relationships between life cycles. However, the case study is still only one, relatively small, example. Therefore, in the conclusions to this chapter, it is briefly explained how the framework of this chapter can be generalized in such a way that it is possible to treat different calls of one method in a different way in the construction of subclasses.

A final note about Definition 4.2.5 on the preceding page is that life-cycle inheritance is defined in terms of a composition of operators and not simply as the disjunction of the two definitions of protocol and projection inheritance. The latter would not be a true combination of protocol and projection inheritance. It would also lack desirable properties such as transitivity.

In reasoning about object life cycles and inheritance relationships between them, it is often convenient to know the set of methods being invoked in a life cycle. For this purpose, the *alphabet* operator is introduced. The alphabet operator yields for each closed $\mathrm{PA}_\delta^\tau+\mathrm{RN}$ term the set of observable actions that the corresponding process in the operational semantics may perform. The alphabet operator is defined inductively using the structure of basic BPA_δ terms, under the assumption that derivability is a congruence for the alphabet operator. In combination with the elimination result for $\mathrm{PA}_\delta^\tau+\mathrm{RN}$ of Property 2.4.39 on Page 48, this assumption means that it is possible to calculate the alphabet of arbitrary closed $\mathrm{PA}_\delta^\tau+\mathrm{RN}$ terms.

Definition 4.2.6. (Alphabet) The alphabet operator $\alpha : \mathcal{C}(\mathrm{PA}_\delta^\tau+\mathrm{RN}) \to \mathcal{P}(E)$ is a function such that, for any closed terms $p, q \in \mathcal{C}(\mathrm{PA}_\delta^\tau+\mathrm{RN})$, $(\mathrm{PA}_\delta^\tau+\mathrm{RN} \vdash p = q) \Rightarrow \alpha(p) = \alpha(q)$. For any $a \in E$ and $p, q \in \mathcal{C}(\mathrm{PA}_\delta^\tau+\mathrm{RN})$, $\alpha(\delta) = \emptyset$, $\alpha(\tau) = \emptyset$, $\alpha(a) = \{a\}$, $\alpha(\tau \cdot p) = \alpha(p)$, $\alpha(a \cdot p) = \{a\} \cup \alpha(p)$, and $\alpha(p + q) = \alpha(p) \cup \alpha(q)$.

Note that it should be verified that the definition of the alphabet operator is consistent with the axioms of set theory. Inconsistencies arise when the combination of the congruence requirement and the inductive definition allows the derivation of an equality between sets that is not derivable from set theory. It is beyond the scope of this thesis to prove that Definition 4.2.6 is consistent with set theory. A detailed study on the alphabet operator in process algebra can be found in (BAETEN, BERGSTRA, AND KLOP, 1987). The alphabet operator is particularly interesting in combination with encapsulation and abstraction. Most of the auxiliary lemmas on encapsulation, abstraction, and the alphabet operator presented in the remainder of this section also appear in (BAETEN, BERGSTRA, AND KLOP, 1987).

Figure 4.2.7 on the following page gives an overview of the four inheritance relations defined above. The arrows depict strict inclusion relations. The correctness of the inclusion relations between protocol/projection inheritance, on the one hand, and protocol and projection inheritance, on the other hand, follows easily from their definitions. The other two inclusion relations follow from the definitions and the following lemma, which implies that encapsulating or hiding the empty set of methods yields the original life cycle.

$$\begin{tikzcd} & \leq_{pp} \arrow[dl] \arrow[dr] & \\ \leq_{pt} \arrow[dr] & & \leq_{pj} \arrow[dl] \\ & \leq_{lc} & \end{tikzcd}$$

Figure 4.2.7: An overview of the four inheritance relations for dynamic behavior.

Lemma 4.2.8. *For any closed term* $p \in \mathcal{C}(\mathrm{PA}_\delta^\tau + \mathrm{RN})$ *and sets* $H, I \subseteq E$,

i) $\alpha(p) \cap H = \emptyset \Rightarrow \mathrm{PA}_\delta^\tau + \mathrm{RN} \vdash \partial_H(p) = p$,

ii) $\alpha(p) \cap I = \emptyset \Rightarrow \mathrm{PA}_\delta^\tau + \mathrm{RN} \vdash \tau_I(p) = p$.

Proof. Structural induction on basic BPA_δ terms. □

By means of a few examples, it is straightforward to show that the inclusion relations in Figure 4.2.7 are indeed strict and that there are no inclusion relations between protocol inheritance and projection inheritance.

Example 4.2.9. Consider the three production units U, U_1, and U_2 introduced in Examples 4.2.2 and 4.2.3 on Page 146 and 4.2.4 on Page 147, respectively. The arguments in the last two of these three examples show that $U_1 \leq_{pt} U$ and $U_2 \leq_{pj} U$. It is not difficult to see that there exist no $I \subseteq E$ such that $\tau_I(U_1) = U$ and no $H \subseteq E$ such that $\partial_H(U_2) = U$. Hence, there are no inclusion relations between protocol and projection inheritance. In addition, it follows that the inclusions between protocol/projection inheritance, on the one hand, and protocol and projection inheritance, on the other hand, are strict.

In order to show that protocol/projection inheritance is not an empty relation, consider the following example. Unit U_3 is a production unit that, as all the units in the other examples, first receives a command. Depending on the command, it continues immediately with its main processing step or it performs a preprocessing step followed by the main processing step. After processing is completed, the processed material is delivered to the environment. The life cycle of U_3 is specified as follows: $\mathit{rcmd} \cdot (\mathit{ppmat} \cdot \mathit{pmat} + \mathit{pmat}) \cdot \mathit{omat}$. Unit U_3 is a subclass under protocol/projection inheritance of unit U of Example 4.2.2 on Page 146. It follows simply from the axioms of $\mathrm{PA}_\delta^\tau + \mathrm{RN}$ that, for $H = I = \{\mathit{ppmat}\}$, $\partial_H(U_3) = \tau_I(U_3) = U$.

Finally, to show that the inclusions between protocol, projection, and life-cycle inheritance are strict, consider a production unit U_4 with life cycle $\mathit{rcmd} \cdot (\mathit{ppmat} \cdot \mathit{pmat} \cdot \mathit{omat} + \mathit{error})$. For $H = \{\mathit{error}\}$ and $I = \{\mathit{ppmat}\}$, it easily follows that $\tau_I \circ \partial_H(U_4) = U$. Hence, $U_4 \leq_{lc} U$. It is not difficult to see that there is no relation between U_4 and U under any of the other inheritance relations.

Figure 4.2.7 shows that life-cycle inheritance is more general than any of the other three inheritance relations. The reason for studying all four relations instead of only life-cycle inheritance is one of separation of concerns. Protocol and projection inheritance each characterize a different type of extension of the behavior of life cycles. In a design process, it can be useful to know in what way a life cycle is extended. In the definition of life-cycle inheritance, the type of extension is lost in generality.

The remainder of this subsection is devoted to studying some basic properties of the four inheritance relations. For protocol, projection, and protocol/projection inheritance, there exists a canonical set of methods that can be encapsulated and/or hidden, namely the set of methods new in the subclass. This conforms to the intuitive definitions of inheritance given earlier. To prove this property, the following two lemmas are needed. They state simple results about encapsulation and abstraction.

Lemma 4.2.10. *For any closed term* $p \in \mathcal{C}(\mathrm{PA}^\tau_\delta{+}\mathrm{RN})$ *and sets* $H, I \subseteq E$*,*

i) $\alpha(\partial_H(p)) \cap H = \emptyset$,

ii) $\alpha(\tau_I(p)) \cap I = \emptyset$.

Proof. Structural induction on basic BPA_δ terms. □

Lemma 4.2.11. *For any closed term* $p \in \mathcal{C}(\mathrm{PA}^\tau_\delta{+}\mathrm{RN})$ *and sets* $H, H', I, I' \subseteq E$*,*

i) $\mathrm{PA}^\tau_\delta{+}\mathrm{RN} \vdash \partial_{H' \cup H}(p) = \partial_{H'} \circ \partial_H(p)$,

ii) $\mathrm{PA}^\tau_\delta{+}\mathrm{RN} \vdash \tau_{I' \cup I}(p) = \tau_{I'} \circ \tau_I(p)$.

Proof. Structural induction on basic BPA_δ terms. □

Lemma 4.2.11 gives versions of Property 2.4.29 on Page 44 for the encapsulation and abstraction operators; Property 2.4.29 gives a general result for arbitrary renaming operators.

Property 4.2.12. *For any object life cycles* $p, q \in \mathcal{C}(\mathrm{PA}^\tau)$*,*

i) $p \le_{pt} q \Leftrightarrow \mathrm{PA}^\tau_\delta{+}\mathrm{RN} \vdash \partial_{\alpha(p)\backslash\alpha(q)}(p) = q$,

ii) $p \le_{pj} q \Leftrightarrow \mathrm{PA}^\tau_\delta{+}\mathrm{RN} \vdash \tau_{\alpha(p)\backslash\alpha(q)}(p) = q$,

iii) $p \le_{pp} q \Leftrightarrow \mathrm{PA}^\tau_\delta{+}\mathrm{RN} \vdash \partial_{\alpha(p)\backslash\alpha(q)}(p) = q \wedge \mathrm{PA}^\tau_\delta{+}\mathrm{RN} \vdash \tau_{\alpha(p)\backslash\alpha(q)}(p) = q$.

Proof. Only the proof of Property *i*) is given. The proof of *ii*) is similar; Property *iii*) follows from *i*) and *ii*).

It follows from Definition 4.2.5 *i*) (Protocol inheritance) on Page 148 that $\mathrm{PA}^\tau_\delta{+}\mathrm{RN} \vdash \partial_{\alpha(p)\backslash\alpha(q)}(p) = q$ implies $p \le_{pt} q$. To prove the other implication, assume $p \le_{pt} q$. It follows from Lemma 4.2.11 *i*) and Lemma 4.2.8 *i*) that there exists a subset H of $\alpha(p)$ such that $\mathrm{PA}^\tau_\delta{+}\mathrm{RN} \vdash \partial_H(p) = q$. The congruence requirement in Definition 4.2.6 on Page 149 for the alphabet operator and Lemma 4.2.10 *i*) show that H cannot contain any methods in $\alpha(q)$. Again, Lemmas 4.2.11 *i*) and 4.2.8 *i*) yield that H can be extended to all elements of $\alpha(p)$ which are not elements of $\alpha(q)$. □

Property 4.2.12 does not have a counterpart for life-cycle inheritance. That is, for object life cycles p and q in $\mathcal{C}(\mathrm{PA}^\tau)$ such that $p \le_{lc} q$, there does not exist a canonical partitioning of $\alpha(p)\backslash\alpha(q)$ into H and I such that $\mathrm{PA}^\tau_\delta{+}\mathrm{RN} \vdash \tau_I \circ \partial_H(p) = q$. Assume, for example, that $p \le_{pp} q$. It follows from Property 4.2.12 *iii*) that for $H = \alpha(p)\backslash\alpha(q)$ and $I = \emptyset$, as well as for $H = \emptyset$ and $I = \alpha(p)\backslash\alpha(q)$, $\mathrm{PA}^\tau_\delta{+}\mathrm{RN} \vdash \tau_I \circ \partial_H(p) = q$.

Two basic properties that any inheritance relation should satisfy are reflexivity and transitivity. It is easy to see that all four inheritance relations are reflexive. Except for life-cycle inheritance, it is also straightforward to show that they are transitive. Showing that life-cycle inheritance is transitive is slightly more involved. The crucial point is the observation that, given two life cycles p and q such that $p \leq_{lc} q$ and $q \leq_{lc} r$, it is not possible that methods in p are encapsulated (hidden) whereas the *same methods* in q are hidden (encapsulated). The reason is simple: Methods of p that are encapsulated or hidden, simply do not occur in q anymore. The following lemmas are needed to formally prove that life-cycle inheritance is transitive.

Lemma 4.2.13. *For any closed term* $p \in \mathcal{C}(\mathrm{PA}_\delta^\tau{+}\mathrm{RN})$ *and sets* $H, I \subseteq E$,

i) $\alpha(\partial_H(p)) \subseteq \alpha(p)$,

ii) $\alpha(\tau_I(p)) \subseteq \alpha(p)$.

Proof. Structural induction on basic BPA_δ terms. □

Lemma 4.2.14. *For any closed term* $p \in \mathcal{C}(\mathrm{PA}_\delta^\tau{+}\mathrm{RN})$ *and sets* $H, I \subseteq E$,
$H \cap I = \emptyset \Rightarrow \mathrm{PA}_\delta^\tau{+}\mathrm{RN} \vdash \tau_I \circ \partial_H(p) = \partial_H \circ \tau_I(p)$.

Proof. Structural induction on basic BPA_δ terms. □

Property 4.2.15. *Protocol, projection, protocol/projection, and life-cycle inheritance are preorders.*

Proof. It follows from Lemma 4.2.8 on Page 150 that, for any $p \in \mathcal{C}(\mathrm{PA}^\tau)$, $\partial_\emptyset(p) = p$ and $\tau_\emptyset(p) = p$. Hence, $\leq_{pt}$, $\leq_{pj}$, $\leq_{pp}$, and $\leq_{lc}$ are reflexive.

To show that $\leq_{pt}$ is transitive, let p, q, and r be object life cycles in $\mathcal{C}(\mathrm{PA}^\tau)$ such that $p \leq_{pt} q$ and $q \leq_{pt} r$. Assume that $H, H' \subseteq E$ are such that $\mathrm{PA}_\delta^\tau{+}\mathrm{RN} \vdash \partial_H(p) = q$ and $\mathrm{PA}_\delta^\tau{+}\mathrm{RN} \vdash \partial_{H'}(q) = r$. Using Lemma 4.2.11 *i*) on the previous page, $\mathrm{PA}_\delta^\tau{+}\mathrm{RN} \vdash \partial_{H' \cup H}(p) = \partial_{H'} \circ \partial_H(p) = \partial_{H'}(q) = r$. Hence, $p \leq_{pt} r$, which proves transitivity of protocol inheritance. The proofs for $\leq_{pj}$ and $\leq_{pp}$ are very similar.

To show that $\leq_{lc}$ is transitive, assume p, q, and r are object life cycles in $\mathcal{C}(\mathrm{PA}^\tau)$ such that $p \leq_{lc} q$ and $q \leq_{lc} r$. Hence, there are subsets H, H', I, and I' of E such that $\mathrm{PA}_\delta^\tau{+}\mathrm{RN} \vdash \tau_I \circ \partial_H(p) = q$ and $\mathrm{PA}_\delta^\tau{+}\mathrm{RN} \vdash \tau_{I'} \circ \partial_{H'}(q) = r$. It follows from the results given so far that these sets can be chosen such that H and I are subsets of $\alpha(p) \backslash \alpha(q)$ and such that H' and I' are subsets of $\alpha(q) \backslash \alpha(r)$ (see also the proof of Property 4.2.12 on the preceding page). The congruence requirement in Definition 4.2.6 on Page 149 for the alphabet operator and Lemma 4.2.13 yield that $\alpha(r) \subseteq \alpha(q) \subseteq \alpha(p)$, and hence that $(H \cup I) \cap (H' \cup I') = \emptyset$. Using Lemmas 4.2.11 and 4.2.14, it follows that $\mathrm{PA}_\delta^\tau{+}\mathrm{RN} \vdash \tau_{I' \cup I} \circ \partial_{H' \cup H}(p) = \tau_{I'} \circ \tau_I \circ \partial_{H'} \circ \partial_H(p) = \tau_{I'} \circ \partial_{H'} \circ \tau_I \circ \partial_H(p) = \tau_{I'} \circ \partial_{H'}(q) = r$. Hence, $p \leq_{lc} r$. □

Any preorder induces an equivalence relation. The meaning of the equivalence relations induced by the inheritance preorders is "subclass equivalence" under the corresponding form of inheritance. Intuitively, two life cycles should be subclass equivalent under any form of inheritance if and only if their equality is derivable from the axioms of PA^τ.

Definition 4.2.16. (Subclass equivalence) Let $\approx_*$, where $* \in \{pp, pt, pj, lc\}$, be the equivalence relation induced by $\leq_*$. That is, for any object life cycles p and q in $\mathcal{C}(\text{PA}^\tau)$, $p \approx_* q \Leftrightarrow p \leq_* q \wedge q \leq_* p$. Processes p and q are said to be *subclass equivalent* under $*$ inheritance.

The four subclass-equivalence relations indeed all coincide with derivability from the axioms of PA^τ. The proof of this claim uses the following conservativity result.

Property 4.2.17. (Conservative extension) *For any closed terms $p, q \in \mathcal{C}(\text{PA}^\tau)$,*
$\text{PA}^\tau_\delta + \text{RN} \vdash p = q \Leftrightarrow \text{PA}^\tau \vdash p = q$.

Proof. Straightforward with the techniques of (BAETEN AND VERHOEF, 1995). □

Property 4.2.18. *For any object life cycles $p, q \in \mathcal{C}(\text{PA}^\tau)$ and $* \in \{pp, pt, pj, lc\}$,*
$p \approx_* q \Leftrightarrow \text{PA}^\tau \vdash p = q$.

Proof. The result is only proven for protocol inheritance. The other proofs are similar. First, assume that $\text{PA}^\tau \vdash p = q$. It follows from Lemma 4.2.8 *i)* on Page 150 and the above conservativity result that $\text{PA}^\tau_\delta + \text{RN} \vdash \partial_\emptyset(p) = p = q$ and $\text{PA}^\tau_\delta + \text{RN} \vdash \partial_\emptyset(q) = q = p$. Hence, $p \leq_{pt} q$ and $q \leq_{pt} p$, which in turn implies that $p \approx_{pt} q$.

Second, assume $p \approx_{pt} q$, which implies that $p \leq_{pt} q$ and $q \leq_{pt} p$. Property 4.2.12 *i)* on Page 151 yields that $\text{PA}^\tau_\delta + \text{RN} \vdash \partial_{\alpha(p)\backslash\alpha(q)}(p) = q$ and $\text{PA}^\tau_\delta + \text{RN} \vdash \partial_{\alpha(q)\backslash\alpha(p)}(q) = p$. It follows from the congruence requirement in Definition 4.2.6 on Page 149 for the alphabet operator and Lemma 4.2.13 *i)* on the preceding page that $\alpha(q) \subseteq \alpha(p)$ and $\alpha(p) \subseteq \alpha(q)$. Hence, $\alpha(p) = \alpha(q)$, which means that $\alpha(p)\backslash\alpha(q) = \alpha(q)\backslash\alpha(p) = \emptyset$. Lemma 4.2.8 *i)* yields that $\text{PA}^\tau_\delta + \text{RN} \vdash \partial_{\alpha(p)\backslash\alpha(q)}(p) = p$ and hence that $\text{PA}^\tau_\delta + \text{RN} \vdash p = q$. Property 4.2.17 implies that $\text{PA}^\tau \vdash p = q$.

Note that since Property 4.2.12 does not have a counterpart for life-cycle inheritance, one has to use the more basic results of the various lemmas given so far to prove the property for life-cycle inheritance. □

Using the above conservativity result and the techniques of (BAETEN AND VERHOEF, 1995), it is possible to prove that the theory PA^τ is complete for rooted branching bisimilarity. As a result, two life cycles are subclass equivalent under any form of inheritance if and only if they are rooted branching bisimilar.

It is important to note that the inheritance preorders are not precongruences for the operators of PA^τ. That is, it is not possible to apply them in arbitrary contexts.

Example 4.2.19. It is easy to see that, for any distinct $a, b \in E$, $a + b \leq_{pt} a$, because $\partial_{\{b\}}(a + b) = a$. However, in a context where an occurrence of b is followed by an a, it is not allowed to replace a by its subclass $a + b$. Doing so, yields $b \cdot (a + b)$. Obviously, since $\partial_{\{b\}}(b \cdot (a + b)) = \delta$, $b \cdot (a + b) \not\leq_{pt} b \cdot a$. A similar example can be constructed for projection inheritance.

Another instructive example is the following, where first a subclass of a single method call a under protocol inheritance is constructed and, subsequently, this subclass is further refined to a more specialized subclass. For any distinct $a, b, c, d \in E$, $a + b \leq_{pt} a$ and

$a \cdot c + b \leq_{pt} b$. Replacing the occurrence of b in the former with its subclass $a \cdot c + b$, yields $a + (a \cdot c + b)$. It is not difficult to see that $a + (a \cdot c + b) \not\leq_{pt} a$. Another subclass of b is $d \cdot c + b$. Replacing b with this subclass, yields $a + (d \cdot c + b)$. It is easy to prove that $a + (d \cdot c + b) \leq_{pt} a$.

These examples show that a problem may arise when a method that is encapsulated or hidden also appears in the context. The fact that the inheritance relations cannot be applied in arbitrary contexts is not really unexpected. The reason is that it is not allowed to treat different calls of the same method in a different way. In the remainder of this subsection, it is formalized under which conditions it is allowed to refine a subclass to a more specialized subclass. Before going into more details, the notion of a context is defined. A context is a closed PA^τ term that contains a "hole."

Definition 4.2.20. (Context) The most simple context is simply a hole, denoted by an underscore "_". Furthermore, for any closed term $p \in \mathcal{C}(\mathrm{PA}^\tau)$ and context $C[\_]$, $p \oplus C[\_]$ and $C[\_] \oplus p$ are contexts, where $\oplus \in \{\cdot, +, \|, \lfloor\!\lfloor\}$.

Example 4.2.21. Let a, b, and c be three identifiers in M. The following are simple examples of contexts: $a + \_$, $\_ \cdot a$, and $a \parallel b \cdot \_$. Substituting c in any of these contexts yields the closed PA^τ terms $a + c$, $c \cdot a$, and $a \parallel b \cdot c$.

Property 4.2.22 given below defines the conditions that must be satisfied when refining a subclass. Let $p, q, r, C[\_], H, H', I,$ and I' be as in Property 4.2.22. Informally, the condition in Property 4.2.22 *i)* that $\alpha(r) \cap H = \emptyset$ means that it is not allowed to encapsulate methods in p that are *not* encapsulated in $C[q]$; methods being encapsulated in p are those in H and methods not encapsulated in $C[q]$ are those in $\alpha(r)$. The conditions in the other properties have similar meanings. The additional requirement in Property *iv)* that $(H' \cup H) \cap (I' \cup I) = \emptyset$ means that the methods being encapsulated must be disjoint from the methods being hidden.

Property 4.2.22. *Let p, q, and r be object life cycles in $\mathcal{C}(\mathrm{PA}^\tau)$ and let $C[\_]$ be a context. Let H, H', I, and I' be subsets of E.*

i) *If $p \leq_{pt} q$, with H such that $\mathrm{PA}^\tau_\delta + \mathrm{RN} \vdash \partial_H(p) = q$, and $C[q] \leq_{pt} r$, then*
$\alpha(r) \cap H = \emptyset \Rightarrow C[p] \leq_{pt} r.$

ii) *If $p \leq_{pj} q$, with I such that $\mathrm{PA}^\tau_\delta + \mathrm{RN} \vdash \tau_I(p) = q$, and $C[q] \leq_{pj} r$, then*
$\alpha(r) \cap I = \emptyset \Rightarrow C[p] \leq_{pj} r.$

iii) *If $p \leq_{pp} q$, with H and I such that $\mathrm{PA}^\tau_\delta + \mathrm{RN} \vdash \partial_H(p) = q$ and $\mathrm{PA}^\tau_\delta + \mathrm{RN} \vdash \tau_I(p) = q$, and if $C[q] \leq_{pp} r$, then*
$\alpha(r) \cap (H \cup I) = \emptyset \Rightarrow C[p] \leq_{pp} r.$

iv) *If $p \leq_{lc} q$ with H and I such that $\mathrm{PA}^\tau_\delta + \mathrm{RN} \vdash \tau_I \circ \partial_H(p) = q$, and if $C[q] \leq_{lc} r$ with H' and I' such that $\mathrm{PA}^\tau_\delta + \mathrm{RN} \vdash \tau_{I'} \circ \partial_{H'}(C[q]) = r$, then*
$(H' \cup H) \cap (I' \cup I) = \emptyset \wedge \alpha(r) \cap (H \cup I) = \emptyset \Rightarrow C[p] \leq_{lc} r.$

Proof. First, Property *i)* is proven. Let $H' \subseteq E$ such that $\mathrm{PA}^\tau_\delta + \mathrm{RN} \vdash \partial_{H'}(C[q]) = r$. A consequence of the axioms of the encapsulation operator is that encapsulation distributes over all operators of PA^{τ}. This result is used in the second step of the following derivation. In the final step, the requirement that $\alpha(r) \cap H = \emptyset$ is used.

$$\partial_{H' \cup H}(C[p]) \overset{4.2.11}{=} \partial_{H'} \circ \partial_H(C[p]) = \partial_{H'} \circ \partial_H(C[\partial_H(p)]) = \partial_{H'} \circ \partial_H(C[q]) \overset{4.2.11}{=}$$
$$\partial_{H' \cup H}(C[q]) = \partial_{H \cup H'}(C[q]) \overset{4.2.11}{=} \partial_H \circ \partial_{H'}(C[q]) = \partial_H(r) \overset{4.2.8}{=} r.$$

This derivation shows that $C[p] \leq_{pt} r$. Properties *ii*) and *iii*) are proven similarly. Property *iv*) is shown as follows. The distributivity of encapsulation and abstraction over all operators of PA^{τ} is used in the third step of the following derivation. The condition that $(H' \cup H) \cap (I' \cup I) = \emptyset$ is used in the second and fifth step. As before, the other condition is used in the final step.

$$\tau_{I' \cup I} \circ \partial_{H' \cup H}(C[p]) \overset{4.2.11}{=} \tau_{I'} \circ \tau_I \circ \partial_{H'} \circ \partial_H(C[p]) \overset{4.2.14}{=} \tau_{I'} \circ \partial_{H'} \circ \tau_I \circ \partial_H(C[p]) =$$
$$\tau_{I'} \circ \partial_{H'} \circ \tau_I \circ \partial_H(C[\tau_I \circ \partial_H(p)] = \tau_{I'} \circ \partial_{H'} \circ \tau_I \circ \partial_H(C[q]) \overset{4.2.11,4.2.14}{=}$$
$$\tau_I \circ \partial_H \circ \tau_{I'} \circ \partial_{H'}(C[q]) = \tau_I \circ \partial_H(r) \overset{4.2.8}{=} r. \qquad \square$$

Note that the proof of Property 4.2.22 uses the fact that encapsulation and abstraction distribute over all operators of PA^{τ}, which includes the left merge and the merge operator. Thus, Property 4.2.22 does not carry over to a process-algebraic setting which allows communication between processes (see also Section 2.4.3, Page 42).

Example 4.2.23. It is easy to check that the examples of Example 4.2.19 on Page 153 are consistent with Property 4.2.22 *i*) on the facing page. The following example shows an application of Property 4.2.22 *iv*). Let $a, b, c \in E$ be distinct methods. Consider the context $C[\_]$ defined as $\_ + b$. By encapsulating method b, it is easy to show that $C[a] \leq_{lc} a$. In addition, $a \cdot c \leq_{lc} a$, which follows from hiding method c. Replacing the occurrence of a in $C[a]$ with its subclass $a \cdot c$, yields $C[a \cdot c]$. In order to apply Property 4.2.22 *iv*), let $H' = \{b\}$, $I = \{c\}$, and $H = I' = \emptyset$. Obviously, this satisfies the requirement that $(H' \cup H) \cap (I' \cup I) = \emptyset$. Since, in addition, $\alpha(a)$ and $H \cup I$ are disjoint, it follows that $C[a \cdot c] \leq_{lc} a$. That is, $a \cdot c + b \leq_{lc} a$. However, $a \cdot b$ is also a subclass of a, which can be easily shown by hiding the singleton $I = \{b\}$. Substituting $a \cdot b$ for a in $C[a]$ yields $C[a \cdot b]$. Since in this case $(H' \cup H) \cap (I' \cup I)$, where H, H', and I' are as before, is not empty, Property 4.2.22 *iv*) cannot be applied. It is not difficult to verify that $C[a \cdot b]$ is not a subclass of a: $C[a \cdot b] \not\leq_{lc} a$. The reason is that in $C[a]$ method b is encapsulated whereas in $a \cdot b$ method b is hidden.

The results presented in this subsection show that the definitions of the four inheritance relations are sound. The conditions under which it is allowed to apply the inheritance preorders in arbitrary contexts or to refine a subclass to a more specialized subclass seem reasonable, although more experience with practical examples has to show whether they are not too restrictive.

4.2.2 Axioms of inheritance

To get a better understanding of the four inheritance relations, it is useful to know what general subclass relations are valid under each form of inheritance. The *axioms of inheritance* given in this subsection show that extending a life cycle with an alternative behavior yields a subclass under protocol inheritance. They also show that extending a life cycle by inserting some behavior in between sequential parts of the original life cycle, as well as putting some behavior in parallel with (part of) the original life cycle yields a subclass under projection inheritance. Life-cycle inheritance allows arbitrary combinations of these three extensions.

Note that no attempt is made to find a complete set of axioms characterizing the inheritance relations. The theory developed in this section is not intended for use in practical applications. The main goal is to learn to understand inheritance of dynamic behavior. The axioms presented in this section are helpful in achieving this goal. In addition, in the next section, they form a source of inspiration for transformation rules in a framework based on Petri nets, which is a framework much closer to the object-oriented methods used in practice than process algebra.

Property 4.2.24. *For any closed terms* $p, q \in \mathcal{C}(\mathrm{PA}^\tau)$ *and method* $b \in E \backslash \alpha(q)$,

$q + b \cdot p \leq_{pt} q$	PT

Proof. Let $H = \{b\}$.
$\partial_H(q + b \cdot p) \overset{D3,D4}{=} \partial_H(q) + \partial_H(b) \cdot \partial_H(p) \overset{D2,4.2.8}{=} q + \delta \cdot \partial_H(p) \overset{A7,A6}{=} q.$
Hence, $q + b \cdot p \leq_{pt} q$. □

Method b functions as some sort of a "*guard*." Blocking the guard means that the environment cannot choose the behavior $b \cdot p$. For this reason, b may not appear in q, since blocking b would otherwise change the behavior of q. Axiom PT shows that encapsulation is sufficient to capture inheritance by means of the choice operator. It also shows that it is sufficient to encapsulate a single method, whereas the canonical set of Property 4.2.12 on Page 151, $\alpha(q + b \cdot p) \backslash \alpha(q)$, might be much larger. Finally, note that Axiom PT has a variant in which the term p is absent. This variant states that it is allowed to extend a life cycle with the alternative of a single method call. This extension is not captured by PT, because the theory PA^τ does not have a means to express the empty process. The subclass relationship between production units U and U_1 discussed in Example 4.2.3 on Page 146 can be proven by means of this variant of PT in combination with the context rule of Property 4.2.22 *i*) on Page 154.

Axiom PT is the most characteristic example of protocol inheritance. Property 4.2.27 given below lists several axioms that are characteristic for projection inheritance. The following two lemmas are needed to prove the correctness of these axioms for projection inheritance.

Lemma 4.2.25. *For any closed term* $p \in \mathcal{C}(\mathrm{PA}^\tau)$, *and* $I \subseteq E$,
$\alpha(p) \subseteq I \Rightarrow \mathrm{PA}_\delta^\tau{+}\mathrm{RN} \vdash \tau_I(p) = \tau$.

Proof. Structural induction on basic BPA terms. □

Note that this lemma cannot be proven for arbitrary closed $\mathrm{PA}_\delta^\tau{+}\mathrm{RN}$ terms, because it is not possible to hide constant δ.

Lemma 4.2.26. *For any closed term* $p \in \mathcal{C}(\mathrm{PA}_\delta^\tau + \mathrm{RN})$,

i) $\mathrm{PA}_\delta^\tau + \mathrm{RN} \vdash p \mathbin{⌊\!⌊} \tau = p$,

ii) $\mathrm{PA}_\delta^\tau + \mathrm{RN} \vdash p \parallel \tau = \tau \cdot p + p$.

Proof. Property *i)* is proven by induction on the structure of basic BPA_δ terms. Property *ii)* follows immediately from *i)* and Axioms $M1$ and $M2$. □

Property 4.2.27. *For any* q, q_0, q_1*, and* r *in* $\mathcal{C}(\mathrm{PA}^\tau)$ *such that* $\alpha(r) \subseteq E \backslash (\alpha(q) \cup \alpha(q_0) \cup \alpha(q_1))$,

$q \cdot r \leq_{pj} q$	$PJ1$
$q \cdot (r \cdot (q_0 + q_1) + q_0) \leq_{pj} q \cdot (q_0 + q_1)$	$PJ2$
$q_0 \cdot (q_1 \parallel r) \leq_{pj} q_0 \cdot q_1$	$PJ3$

Proof. Let I be equal to $\alpha(r)$.

$\tau_I(q \cdot r) \overset{TI4}{=} \tau_I(q) \cdot \tau_I(r) \overset{4.2.8}{=} q \cdot \tau_I(r) \overset{4.2.25}{=} q \cdot \tau \overset{B1}{=} q$,

which proves $PJ1$. The proof for $PJ2$ is similar, only Axiom $B2$ is used instead of $B1$.

$\tau_I(q \cdot (r \cdot (q_0 + q_1) + q_0)) \overset{TI3,TI4}{=} \tau_I(q) \cdot (\tau_I(r) \cdot (\tau_I(q_0) + \tau_I(q_1)) + \tau_I(q_0)) \overset{4.2.8,4.2.25}{=}$
$q \cdot (\tau \cdot (q_0 + q_1) + q_0) \overset{B2}{=} q \cdot (q_0 + q_1)$.

Axiom $PJ3$ is proved as follows.

$\tau_I(q_0 \cdot (q_1 \parallel r)) \overset{TI4,TIM1}{=} \tau_I(q_0) \cdot (\tau_I(q_1) \parallel \tau_I(r)) \overset{4.2.8,4.2.25}{=} q_0 \cdot (q_1 \parallel \tau) \overset{4.2.26}{=}$
$q_0 \cdot (\tau \cdot q_1 + q_1) \overset{A6,B2}{=} q_0 \cdot q_1$. □

Axioms $PJ1$ and $PJ2$ are inspired by the two Axioms $B1$ and $B2$. Together they state that inserting new behavior in an object life cycle that does not disable any behavior of the original life cycle yields a subclass under projection inheritance. Axiom $PJ3$ shows that putting alternative behavior in parallel with part of the original life cycle also yields a subclass under projection inheritance. The inheritance relationship between production units U and U_2 discussed in Example 4.2.4 on Page 147 follows from Axiom $PJ1$ and the context rule of Property 4.2.22 *ii)* on Page 154.

Example 4.2.28. Consider production unit U of Example 4.2.2 on Page 146. Production unit U_5 sends a processing-ready signal when the main processing step is completed. The signal is sent in parallel with the delivery of output material. The life cycle of U_5 is described as follows: $rcmd \cdot pmat \cdot (omat \parallel sprdy)$. It follows from Axiom $PJ3$ that $U_5 \leq_{pj} U$.

Property 4.2.29. *For any* q_0, q_1*, and* r *in* $\mathcal{C}(\mathrm{PA}^\tau)$ *such that* $\alpha(r) \subseteq E \backslash (\alpha(q_0) \cup \alpha(q_1))$*, and* $b \in E \backslash (\alpha(q_0) \cup \alpha(q_1))$,

$q_0 \cdot (b \cdot r \cdot q_1 + q_1) \leq_{pp} q_0 \cdot q_1$	PP

Proof. Let H be equal to $\{b\}$ and I be equal to $\alpha(r) \cup \{b\}$. Note that b is not equal to τ.

$\partial_H(q_0 \cdot (b \cdot r \cdot q_1 + q_1)) \overset{D4,D3}{=} \partial_H(q_0) \cdot (\partial_H(b) \cdot \partial_H(r \cdot q_1) + \partial_H(q_1)) \overset{D2,4.2.8}{=}$
$q_0 \cdot (\delta \cdot \partial_H(r \cdot q_1) + q_1) \overset{A7,A6}{=} q_0 \cdot q_1$.

Furthermore,

$$\tau_I(q_0 \cdot (b \cdot r \cdot q_1 + q_1)) \overset{TI4,TI3}{=} \tau_I(q_0) \cdot (\tau_I(b \cdot r) \cdot \tau_I(q_1) + \tau_I(q_1)) \overset{4.2.8,4.2.25}{=}$$
$$q_0 \cdot (\tau \cdot q_1 + q_1) \overset{A6,B2}{=} q_0 \cdot q_1.$$

It follows from the above two derivations that $q_0 \cdot ((b \cdot r) \cdot q_1 + q_1) \leq_{pp} q_0 \cdot q_1$. □

Axiom *PP* shows that under protocol/projection inheritance it is allowed to *postpone* behavior. In case it is possible to specify recursive behavior, a nice variant of *PP* is an axiom in which the behavior $b \cdot r$ is iterated arbitrarily many times before continuing with q_1. The inheritance relationship between production units U and U_3 shown in Example 4.2.9 on Page 150 can also be proven by means of a variant of Axiom *PP* without term r in combination with the context rule of Property 4.2.22 *iii*) on Page 154.

The final property of this section gives a few axioms of life-cycle inheritance.

Property 4.2.30. *For any* p, q_0, q_1, *and* r *in* $\mathcal{C}(\mathrm{PA}^\tau)$ *such that* $\alpha(r) \subseteq E\backslash(\alpha(q_0) \cup \alpha(q_1))$, *and* $b \in E\backslash(\alpha(r) \cup \alpha(q_0) \cup \alpha(q_1))$,

$q_0 \cdot (r \cdot (q_1 + b \cdot p)) \leq_{lc} q_0 \cdot q_1$	*LC*1
$q_0 \cdot ((q_1 + b \cdot p) \parallel r) \leq_{lc} q_0 \cdot q_1$	*LC*2
$q_0 \cdot (q_1 \parallel (r + b \cdot p)) \leq_{lc} q_0 \cdot q_1$	*LC*3

Proof. Let H be equal to $\{b\}$ and I be equal to $\alpha(r)$.

$$\tau_I \circ \partial_H(q_0 \cdot (r \cdot (q_1 + b \cdot p))) \overset{D4,D3,TI4,TI3}{=}$$
$$\tau_I \circ \partial_H(q_0) \cdot (\tau_I \circ \partial_H(r) \cdot (\tau_I \circ \partial_H(q_1) + \tau_I \circ \partial_H(b) \cdot \tau_I \circ \partial_H(p))) \overset{D2,4.2.8,4.2.25}{=}$$
$$q_0 \cdot (\tau \cdot (q_1 + \delta \cdot \tau_I \circ \partial_H(p))) \overset{A7,A6,B1}{=} q_0 \cdot q_1,$$

which proves *LC*1. Axiom *LC*2 is proven as follows:

$$\tau_I \circ \partial_H(q_0 \cdot ((q_1 + b \cdot p) \parallel r)) \overset{D4,DM1,D3}{=} \tau_I(\partial_H(q_0) \cdot (\partial_H(q_1) + \partial_H(b) \cdot \partial_H(p)) \parallel \partial_H(r)))$$
$$\overset{D2,4.2.8}{=} \tau_I(q_0 \cdot ((q_1 + \delta \cdot \partial_H(p)) \parallel r)) \overset{A7,A6}{=} \tau_I(q_0 \cdot (q_1 \parallel r)),$$

after which the proof proceeds as for Axiom *PJ*3. Axiom *LC*3 is proven similarly. □

Axioms *LC*1 through 3 are combinations of the axioms for protocol and projection inheritance. They are illustrative for life-cycle inheritance. It is not difficult to find several more of such combinations. It is possible to prove the above axioms by means of the context rule of Property 4.2.22 on Page 154. The proof above uses basic lemmas and axioms, because they yield a more readable proof.

Example 4.2.31. Consider a production unit U_6 with life cycle $rcmd \cdot ((pmat + ppmat \cdot pmat) \parallel ssps) \cdot omat$. Upon receipt of a command, unit U_6 sends a start-processing signal. In parallel, it starts its main processing step, possibly preceded by a preprocessing step. It follows from Axiom *LC*2 and the context rule of Property 4.2.22 *iv*) on Page 154 that U_6 is a subclass under life-cycle inheritance of unit U of Example 4.2.2 on Page 146.

In conclusion, this section presents a characterization of inheritance of dynamic behavior in a simple process-algebraic setting. Four inheritance relations are defined, all in terms of the algebraic concepts of encapsulation and abstraction. Several basic properties of the relations show that they are theoretically sound. The axioms of inheritance given in the last subsection show that life-cycle inheritance captures three basic operators to construct a subclass from a superclass, namely choice, sequential composition, and parallel composition. Since these three operators are fundamental to any (concurrent) object-oriented language, it appears that encapsulation and abstraction capture the essence of inheritance of dynamic behavior. The axioms of inheritance show how to create subclasses of some given class in a *constructive* way at design time. Thus, the axioms stimulate the *reuse* of object life cycles during the design process.

4.3 A Petri-Net-Based Approach

The main goal of this section is to translate the results of the previous section to a framework that is close to practical object-oriented methods as OMT, OOD, and UML. Petri nets are well suited for this purpose for several reasons. First, they provide a graphical description technique that is easy to understand and close to state-transition diagrams. Second, Petri nets have an explicit representation of states. Third, it is natural to model concurrency in Petri nets, which is an advantage when modeling distributed systems. Finally, they have a sound theoretical basis and many techniques and tools are available for the analysis of Petri nets.

The results presented in this section are based on (VAN DER AALST AND BASTEN, 1997). In the next subsection, the notion of an object life cycle is formalized in the framework of Petri nets. In Section 4.3.2, the four inheritance relations defined in the previous section are translated to this framework. In Section 4.3.3, several transformation rules on object life cycles are presented. These transformation rules are based on the axioms of inheritance of the previous section. They can be used to transform a class into a subclass, thus, reusing life-cycle designs.

The Petri-net framework of this section is more expressive and more powerful than the process-algebraic framework of the previous section. The price to be paid is that the definitions, the theorems, and the proofs are more complex. The translation from process algebra to Petri nets shows how the development of a concept in one formalism can inspire the development in another formalism. The formalization of inheritance of behavior in process algebra has led to a clear conceptual understanding; the study in Petri nets yields a framework close to practical object-oriented methods.

4.3.1 Object life cycles

An object life cycle specifies the order in which the methods of an object may be executed. When modeling a life cycle with a Petri net, a transition firing corresponds to the execution of a method. Since the emphasis is on the execution order of methods and not on their implementation details, the uncolored formalism of labeled P/T nets as introduced in Chapter 2 is well suited as the basic framework for modeling life cycles and studying inheritance of dynamic behavior. Transition labels correspond to method identifiers.

As in the previous section, the set of methods is denoted M. Recall that M includes the identifier τ to model internal methods. The set of external methods $M\backslash\{\tau\}$ is denoted

E. For the sake of simplicity, it is assumed that all arc weights are one. This means that the class of P/T nets used to define object life cycles is the class of M-labeled, ordinary P/T nets, as introduced in Section 2.3.4. However, not every M-labeled, ordinary P/T net is an object life cycle.

A life cycle refers to a *single object*. It suffices to consider just one object, because multiple objects of the same class interact via the execution of methods and not directly via the life cycle. A P/T net defining a life cycle has exactly one *initial* or *input* place i. Place i has no input transitions and it is the only place marked in the initial marking. A token in place i models the fact that the corresponding object has not yet been created. Because of the single-object requirement, place i contains only a single token in the initial marking.

To model object termination, an object life cycle has a unique *final* or *output* place o. An object terminates when, and only when, it reaches the marking of a single token in o. In addition, if a marking has a token in o, it must be the only token in the marking. This means that, upon termination of an object, all information about the object is removed. Furthermore, it is assumed that it is always possible to terminate. However, this does not mean that an object is *forced* to terminate. In other words, marking $[o]$ must be a home marking of the P/T net modeling an object life cycle, as defined in Definition 2.3.31 on Page 25.

In addition to the above requirements, a P/T net modeling a life cycle is assumed to be connected, as defined in Definition 2.3.15 on Page 21. Furthermore, a life cycle may not have any dead transitions, as defined in Definition 2.3.26 on Page 24. These requirements are technically convenient. They are also meaningful from a design point of view. Places in a part of the P/T net that is not connected to the initial place will remain unmarked no matter what transitions are fired. Dead transitions correspond to methods that cannot be executed. Hence, it is not meaningful to model a life cycle with an unconnected P/T net or a P/T net with dead transitions.

The above considerations are partly related to the static structure of an object life cycle and partly to its dynamic behavior. Therefore, first, the operational semantics for M-labeled, ordinary P/T nets is defined. This semantics takes into account the presence of silent actions. It also formalizes the termination behavior of marked nets modeling life cycles sketched above. Second, it is defined when an M-labeled, ordinary P/T net is an object life cycle. Assume that the universe of identifiers U contains the special identifiers i and o.

Definition 4.3.1. (Operational semantics) The operational semantics $\mathcal{S}^{\mathcal{O}}$ for M-labeled, ordinary P/T nets is the process space $(\mathcal{O}, M, [\,\rangle, \downarrow)$ with branching bisimilarity, $\sim_b$, as the equivalence relation. The set of processes $\mathcal{O}$ is the set of all marked, M-labeled, ordinary P/T nets. The transition relation $[\,\rangle$ is the firing rule of Definition 2.3.7 on Page 18 instantiated with the labeling action function of Definition 2.3.8 on Page 19. Predicate $\downarrow$ is the set of all marked nets $(N, [o])$ in $\mathcal{O}$, where o is the special output place in U introduced above.

The semantics of Definition 4.3.1 uses branching bisimilarity as the equivalence relation, whereas in the previous section rooted branching bisimilarity was used. As explained in Chapter 2, the root condition is crucial in a process-algebraic context. However, it is not needed in a framework of P/T nets. On the contrary, it is often tedious to have to take into account the root condition. Therefore, in this section, branching bisimilarity is chosen as the basic equivalence. However, all the results in this section carry over to rooted branching bisimilarity.

The definition of an object life cycle uses Definition 2.3.17 on Page 21 that defines the set of reachable markings of a P/T net.

Definition 4.3.2. (Object life cycle) Let (N, s) be a marked, M-labeled, ordinary P/T net in $\mathcal{O}$, where N equals (P, T, F, ℓ). Net (N, s) is an *object life cycle* if and only if the following conditions are satisfied:

i) *Connectedness*: N is weakly connected;

ii) *Object creation*: P contains an input place $i \in U$ such that $^\bullet i = \emptyset$;

iii) *Object termination*: P contains an output place $o \in U$ such that $o^\bullet = \emptyset$;

iv) *Single-object requirement*: $s = [i]$;

v) *Proper termination*: for any reachable marking $s' \in [N, s\rangle$, if $o \in s'$, then $s' = [o]$;

vi) *Termination option*: marking $[o]$ is a home marking of (N, s);

vii) *Dead transitions*: (N, s) contains no dead transitions.

The set of all object life cycles is denoted $\mathcal{L}$.

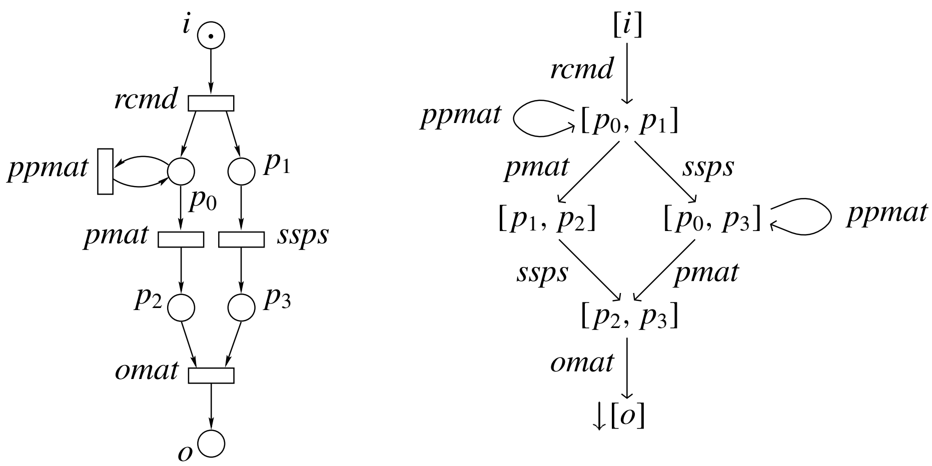


Figure 4.3.3: An example of an object life cycle.

Example 4.3.4. Figure 4.3.3 shows an example of a labeled, ordinary P/T net modeling the object life cycle of a production unit and its operational semantics. The firing rule is depicted by arrows. The states in the semantics are represented by the markings of the net.

The life cycle shows concurrent as well as iterative behavior. After receiving a command, the unit starts its processing phase. In parallel, it sends a start-processing signal to the operator. The processing phase consists of zero or more preprocessing steps, followed by the main processing action. The unit finishes with delivering the output material.

It is clear that the object life cycle of Figure 4.3.3 satisfies the requirements of Definition 4.3.2 on the previous page. In particular, it has always the option to terminate and termination is always proper.

Definition 4.3.2 of an object life cycle in the framework of P/T nets is more involved than the corresponding definition in the process algebra PA^τ, Definition 4.2.1 on Page 146. The main reason for this difference is that the process algebra PA^τ does not contain the inaction constant nor does it allow the specification of iterative behavior. A consequence of these limitations is that the termination of an object life cycle specified in PA^τ is always guaranteed. Labeled, ordinary P/T nets, on the other hand, inherently allow iterations, as the above example shows. It is also not difficult to give P/T nets which will never terminate due to a deadlock. In other words, labeled, ordinary P/T nets are more expressive than the process algebra PA^τ. One of the consequences is that it is necessary to explicitly define the termination requirements in the definition of an object life cycle in the P/T-net framework of this section.

An interesting question is what properties object life cycles exhibit. An example of a useful property is the following.

Property 4.3.5. *An object life cycle $(N, [i]) \in \mathcal{L}$ is bounded.*

Proof. Assume $(N, [i])$ is unbounded. According to Property 2.3.23 (Characterization of boundedness) on Page 23, there exist markings $s' \in [N, [i]\rangle$ and $s'' \in [N, s'\rangle$ such that $s'' > s'$. It follows from Requirement *vi*) (Termination option) of Definition 4.3.2 (Object life cycle) on the previous page that $[o] \in [N, s'\rangle$. Property 2.3.21 (Monotonicity of reachable markings) on Page 22 yields that $[o] \uplus (s'' - s') \in [N, s''\rangle$. Since $s'' - s'$ is not empty, the latter contradicts Requirement *v*) (Proper termination) of Definition 4.3.2 (Object life cycle). Hence, $(N, [i])$ is bounded. □

This property is one step towards a very interesting result, namely that object life cycles can be characterized in terms of liveness and boundedness. The characterization is taken from (VAN DER AALST, 1997), where it is given for so-called sound workflow nets. Sound workflow nets are almost identical to object life cycles. The following auxiliary definition is needed to formulate the desired theorem. Given a net with an input place and an output place as in Definition 4.3.2 (Object life cycle) on the preceding page, it defines the extension of the net with an extra transition connecting the output place to the input place.

Definition 4.3.6. Let $N = (P, T, F, \ell)$ be a labeled, ordinary P/T net satisfying the first three requirements of an object life cycle given in Definition 4.3.2 on the previous page. Assume that $\bar{t}$ is an identifier in U that does not appear in P or T. The labeled, ordinary P/T net $\bar{N} = (\bar{P}, \bar{T}, \bar{F}, \bar{\ell})$ is defined as follows: $\bar{P} = P$, $\bar{T} = T \cup \{\bar{t}\}$, $\bar{F} = F \cup \{(o, \bar{t}), (\bar{t}, i)\}$, and $\bar{\ell} = \ell \cup \{(\bar{t}, \tau)\}$.

The label of the new transition in the extended net does not play a role in the characterization of object life cycles. For the sake of convenience, the label is set to τ.

Theorem 4.3.7. (Characterization of object life cycles) *Let (N, s) be a labeled, ordinary P/T net in $\mathcal{O}$ satisfying Requirements $i)$ through $iv)$ of an object life cycle given in Definition 4.3.2 on Page 161. Net (N, s) is an object life cycle if and only if $(\bar{N}, s)$ is live and bounded.*

Proof. The proof is identical to the proof of Theorem 11 in (VAN DER AALST, 1997), which characterizes soundness of workflow nets in terms of liveness and boundedness. There are two small differences between the two theorems. First, object life cycles are labeled nets whereas workflow nets are unlabeled. However, labeling does not play a role in the proof. Second, the definition of workflow nets requires that the extension with an extra transition connecting the output place to the input place is strongly connected. However, this extra requirement is not needed in the proof. □

Another interesting question is whether it is possible to decide efficiently whether a labeled, ordinary P/T net is an object life cycle.

Theorem 4.3.8. (Decidability of object-life-cycle properties) *Let (N, s) be a labeled, ordinary P/T net in $\mathcal{O}$. It is decidable whether (N, s) is an object life cycle.*

Proof. Requirements *i*) through *iv*) of Definition 4.3.2 on Page 161 are simple structural properties that can be checked for any ordinary P/T net in a straightforward way. Requirements *v*), *vi*), and *vii*) can be derived from the coverability tree of (N, s) (see PETERSON, 1981, where it is called the reachability tree). Since the coverability tree of an ordinary P/T net is always finite, it is decidable whether (N, s) is an object life cycle. □

The exact complexity of deciding the object-life-cycle properties is still an open question. However, some known complexity results for P/T nets (see, for example, ESPARZA AND NIELSEN, 1994) suggest that they cannot be decided efficiently. First, deciding the life-cycle properties by means of the coverability tree, as suggested in the proof of Theorem 4.3.8, requires in the worst case non-primitive recursive space. Second, Requirements *v*), *vi*), and *vii*) of Definition 4.3.2 on Page 161 are all closely related to the question whether a marking is reachable from some other marking. This question is decidable, but the exact complexity is unknown. It is only known that the space requirements of an algorithm to decide reachability are at least exponential in the number of nodes of the net, which is not particularly efficient. The question whether some given marking is a home marking is decidable, but the complexity is unknown. Third, Theorem 4.3.7 shows that object life cycles can be characterized in terms of liveness and boundedness. Deciding boundedness for labeled, ordinary P/T nets is known to be EXPSPACE-hard. The complexity of deciding liveness is unknown. It is unlikely that deciding the combination of liveness and boundedness requires less than exponential space.

Although the complexity of deciding the life-cycle properties is discouraging, it is still possible to verify the properties by means of the coverability tree in a relatively straightforward way for object life cycles that are not too large. In addition, (KOVALYOV, 1996) gives conditions for a marked, ordinary P/T net which guarantee that the net is live and bounded. It also presents an algorithm which is linear in the number of nodes of the net to verify these conditions. Based on Theorem 4.3.7, the algorithm of (KOVALYOV, 1996) provides an efficient procedure to prove that a P/T net is an object life cycle, although failure of the algorithm does not mean that the net is not a life cycle. Experience with case studies is needed to find out the practical limitations. In case the P/T nets in an object-oriented design become too complex for the automatic verification of life-cycle properties or if the algorithm of (KOVALYOV, 1996) fails, there are two solutions. First, it is possible to build life cycles in a constructive way by means of transformation rules preserving the life-cycle properties. Second, it is possible to restrict the class of P/T nets used in the design in such a way that the verification of the life-cycle properties becomes more efficient. Both solutions are discussed briefly in the remainder of this subsection.

The characterization of object life cycles in terms of liveness and boundedness of Theorem 4.3.7 already provides the basis for a useful set of transformation rules preserving life-cycle properties. The theorem implies that transformation rules preserving liveness and boundedness of ordinary P/T nets also preserve life-cycle properties. Starting with an object life cycle, the application of any such transformation rule yields another object life cycle. Using this approach, time-consuming verification of life-cycle properties is not necessary. The literature contains several studies on transformation rules preserving liveness and boundedness of ordinary P/T nets. In (VAN DER AALST, 1997), a set of transformation rules is given that is designed for the purpose of constructing sound workflow nets. They cover design constructs such as the addition of sequential, alternative, parallel, and iterative behavior to a workflow net. Since these rules preserve liveness and boundedness, they can also be used to construct object life cycles. In (VALETTE, 1979), a very similar set of transformations preserving liveness and boundedness of P/T nets is presented, although the context differs from the context of (VAN DER AALST, 1997). Other liveness-and-boundedness-preserving transformation rules can be found in (BERTHELOT, 1987; DESEL, 1990; DESEL AND ESPARZA, 1995; ESPARZA, 1990; KOVALYOV, 1990; MURATA, 1989). However, these rules are developed for the analysis of net models instead of their construction. Consequently, from the design point of view, they do not always have an intuitive meaning. Of course, this does not mean that they cannot be used during the construction of an object-oriented system design. Since the literature already contains so many studies on transformation rules preserving liveness and boundedness of ordinary P/T nets, they are not further discussed in this thesis.

Another source of transformation rules preserving object-life-cycle properties are the transformation rules introduced in Section 4.3.3, which are developed for the purpose of constructing *subclasses* of object life cycles. For more details, the reader is referred to Section 4.3.3.

As mentioned, another solution to make the automatic verification of life-cycle properties feasible is to restrict the class of P/T nets allowed in the design. In Section 2.3.4, the set of free-choice P/T nets has been introduced as a class of nets that combines expressiveness with strong analysis techniques. It is known that deciding the combination of liveness and boundedness for marked, free-choice P/T nets can be done in linear time (KOVALYOV, 1996). In combination with Theorem 4.3.7 on Page 163, this leads immediately to the following result.

Theorem 4.3.9. (Complexity of deciding life-cycle properties for free-choice P/T nets) *For any labeled, free-choice P/T net* (N, s) *in* $\mathcal{O}$, *it is possible to decide the object-life-cycle properties of Definition 4.3.2 on Page 161 in linear time.*

Proof. Theorem 4.3.7 on Page 163 and the results of (KOVALYOV, 1996). □

This last theorem is a small improvement of a result in (VAN DER AALST, 1997) for free-choice workflow nets. It is possible to relax the free-choice requirement in Theorem 4.3.9 slightly while maintaining the same efficiency in deciding the life-cycle properties (see VAN DER AALST, 1997).

Another class of P/T nets that might be an interesting candidate for defining object life cycles is the class of well-handled P/T nets, as defined in (VAN DER AALST, 1996, 1998). For well-handled P/T nets, it is also possible to verify the object-life-cycle properties efficiently. As for free-choice P/T nets, this result is based on Theorem 4.3.7 (Characterization of object life cycles) on Page 163. For more details, the interested reader is referred to (VAN DER AALST, 1996, 1998).

4.3.2 Inheritance relations

Definition 4.3.2 on Page 161 formalizes the notion of an object life cycle in terms of P/T nets. The next step in the translation of the process-algebraic framework of Section 4.2 to P/T nets is the formalization of the four inheritance relations of Definition 4.2.5 on Page 148. Recall that these relations are all defined in terms of two operators, namely encapsulation and abstraction. Encapsulation is used to block method calls; abstraction is used to hide method calls. These two operators can be defined on labeled, ordinary P/T nets as follows. Note that they only affect the structure of a P/T net. For the sake of readability, they are defined on marked nets.

Definition 4.3.10. (Encapsulation) Let (N, s) be a marked, M-labeled, ordinary P/T net in $\mathcal{O}$, where $N = (P, T_0, F_0, \ell_0)$. For any $H \subseteq E$, the encapsulation operator $\partial_H : \mathcal{O} \to \mathcal{O}$ is a function that removes from a given P/T net all transitions with a label in H. Formally, $\partial_H(N, s) = ((P, T_1, F_1, \ell_1), s)$ such that $T_1 = \{t \in T_0 \mid \ell_0(t) \notin H\}$, $F_1 = F_0 \cap ((P \times T_1) \cup (T_1 \times P))$, and $\ell_1 = \ell_0 \cap (T_1 \times M)$.

Definition 4.3.11. (Abstraction) Let (N, s) be a marked, M-labeled, ordinary P/T net in $\mathcal{O}$, with $N = (P, T, F, \ell_0)$. For any $I \subseteq E$, the abstraction operator $\tau_I : \mathcal{O} \to \mathcal{O}$ is a function that renames all transition labels in I to the silent action τ. Formally, $\tau_I(N, s) = ((P, T, F, \ell_1), s)$ such that, for any $t \in T$, $\ell_0(t) \in I$ implies $\ell_1(t) = \tau$ and $\ell_0(t) \notin I$ implies $\ell_1(t) = \ell_0(t)$.

The following property shows that the definition of encapsulation and abstraction is sound. It states that two nets with the same behavior also exhibit the same behavior after encapsulating or abstracting one or more actions.

Property 4.3.12. *Branching bisimilarity,* $\sim_b$, *is a congruence for the encapsulation and abstraction operators.*

Proof. Theorem 2.2.25 on Page 15 proves that branching bisimilarity is an equivalence relation. It remains to be shown that, for any two marked, ordinary P/T nets (N_0, s_0) and (N_1, s_1) in $\mathcal{O}$ and any sets $H, I \subseteq E$, $(N_0, s_0) \sim_b (N_1, s_1)$ implies that $\partial_H(N_0, s_0) \sim_b \partial_H(N_1, s_1)$ and $\tau_I(N_0, s_0) \sim_b \tau_I(N_1, s_1)$. Let $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{O}$ be a branching bisimulation between (N_0, s_0) and (N_1, s_1). Based on $\mathcal{R}$, a relation $\mathcal{Q} \subseteq \mathcal{O} \times \mathcal{O}$ is defined as the set $\{(\partial_H(N_0, u), \partial_H(N_1, v)) \mid (N_0, u)\mathcal{R}(N_1, v)\}$. It is not difficult to verify that $\mathcal{Q}$ is a branching bisimulation between (N_0, s_0) and (N_1, s_1). Hence, $\sim_b$ is a congruence for the encapsulation operator ∂_H. In a similar way, it can be shown that $\sim_b$ is a congruence for the abstraction operator τ_I. $\square$

The introduction of encapsulation and abstraction on P/T nets is sufficient to translate the inheritance relations of Definition 4.2.5 on Page 148 to the framework of this section.

Definition 4.3.13. (Inheritance relations)

i) *Protocol inheritance*:
For any object life cycles $(N_0, [i])$ and $(N_1, [i])$ in $\mathcal{L}$, life cycle $(N_1, [i])$ is a subclass of $(N_0, [i])$ under protocol inheritance, denoted $(N_1, [i]) \leq_{pt} (N_0, [i])$, if and only if there is an $H \subseteq E$ such that $\partial_H(N_1, [i]) \sim_b (N_0, [i])$.

ii) *Projection inheritance*:
For any object life cycles $(N_0, [i])$ and $(N_1, [i])$ in $\mathcal{L}$, life cycle $(N_1, [i])$ is a subclass of $(N_0, [i])$ under projection inheritance, denoted $(N_1, [i]) \leq_{pj} (N_0, [i])$, if and only if there is an $I \subseteq E$ such that $\tau_I(N_1, [i]) \sim_b (N_0, [i])$.

iii) *Protocol/projection inheritance*:
For any object life cycles $(N_0, [i])$ and $(N_1, [i])$ in $\mathcal{L}$, life cycle $(N_1, [i])$ is a subclass of $(N_0, [i])$ under protocol/projection inheritance, denoted $(N_1, [i]) \leq_{pp} (N_0, [i])$, if and only if there is an $H \subseteq E$ such that $\partial_H(N_1, [i]) \sim_b (N_0, [i])$ *and* an $I \subseteq E$ such that $\tau_I(N_1, [i]) \sim_b (N_0, [i])$.

iv) *Life-cycle inheritance*:
For any object life cycles $(N_0, [i])$ and $(N_1, [i])$ in $\mathcal{L}$, life cycle $(N_1, [i])$ is a subclass of $(N_0, [i])$ under life-cycle inheritance, denoted $(N_1, [i]) \leq_{lc} (N_0, [i])$, if and only if there are an $I \subseteq E$ and an $H \subseteq E$ such that $I \cap H = \emptyset$ and $\tau_I \circ \partial_H(N_1, [i]) \sim_b (N_0, [i])$.

Example 4.3.15. Figure 4.3.14 on the next page shows five object life cycles, each of them modeling a variant of a production unit. Unit $(N_0, [i])$ is the basic production unit that receives a command, processes material, and delivers output material. Unit $(N_1, [i])$ extends $(N_0, [i])$ with error control. If upon completion of the processing phase an error is detected, method *repp* is executed to repeat the processing step. By encapsulating method *repp*, it is easily shown that unit $(N_1, [i])$ is a subclass of unit $(N_0, [i])$ under protocol inheritance. Unit $(N_2, [i])$ introduces a new method *cerr* that an operator may use to correct an error. As unit $(N_1, [i])$, unit $(N_2, [i])$ is a subclass under protocol inheritance of unit $(N_0, [i])$. Again, encapsulating method *repp* is sufficient to prove this relationship. Unit $(N_2, [i])$ is a subclass under projection inheritance of unit $(N_1, [i])$. This can be shown

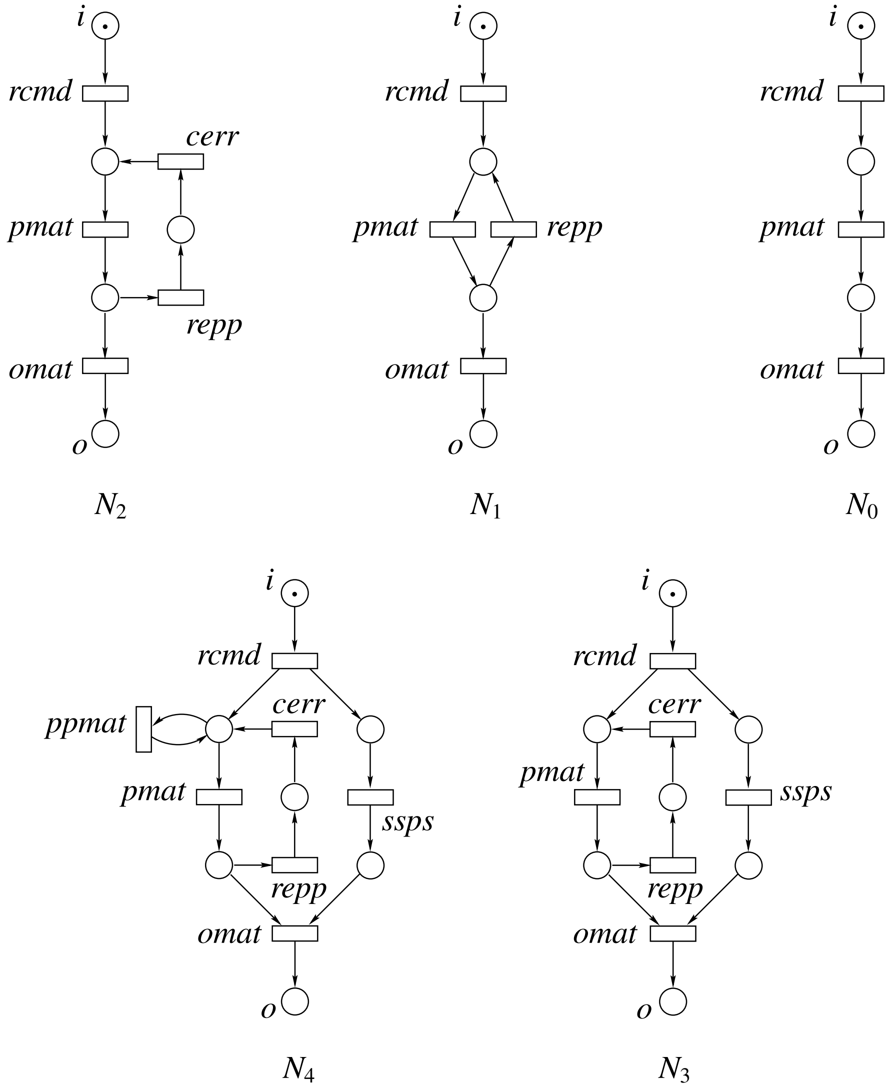


Figure 4.3.14: Some examples of subclasses.

by hiding method *cerr*. Unit $(N_3, [i])$ sends a start-processing signal to the operator in parallel with its processing phase. Hiding method *ssps* shows that it is a subclass under projection inheritance of unit $(N_2, [i])$. Finally, unit $(N_4, [i])$ is a subclass under protocol/projection inheritance of unit $(N_3, [i])$. Encapsulating the preprocessing action *ppmat* yields a life cycle identical to unit $(N_3, [i])$. Hiding method *ppmat* yields a life cycle which is branching bisimilar to $(N_3, [i])$. Finally, all units $(N_j, [i])$, where $0 < j \leq 4$, are subclasses of unit $(N_0, [i])$ under life-cycle inheritance.

In the process-algebraic framework of the previous section, several properties of the four inheritance relations have been proven. Most of these properties carry over to the setting of this section. To facilitate reasoning about life cycles, the alphabet operator is defined on P/T nets.

Definition 4.3.16. (Alphabet) The alphabet operator is a function $\alpha : \mathcal{O} \to \mathcal{P}(E)$. Let (N, s) be a marked, M-labeled, ordinary P/T net in $\mathcal{O}$, with $N = (P, T, F, \ell)$. The alphabet of (N, s) is defined as the set of visible labels of all transitions of the net that are not dead: $\alpha(N, s) = \{\ell(t) \mid t \in T \wedge \ell(t) \neq \tau \wedge t \text{ is not dead}\}$.

An important property of the alphabet operator is that two equivalent P/T nets have the same alphabet. To prove this property, the following lemma is needed. For every two processes related by some branching bisimulation in a given process space, it says that the branching bisimulation relates any process reachable from one of these two processes to a process reachable from the other process.

Lemma 4.3.17. *Assume that $(\mathcal{P}, \mathcal{A}, \longrightarrow, \downarrow)$ is a process space as defined in Definition 2.2.1 on Page 6. Furthermore, assume that $\_ \stackrel{*}{\Longrightarrow} \_ \subseteq \mathcal{P} \times \mathcal{P}$ is the reachability relation of Definition 2.2.2 also on Page 6. Let p and q be two processes in $\mathcal{P}$; finally, let $\mathcal{R}$ be a branching bisimulation between p and q, as defined in Definition 2.2.15 on Page 11. For any $p', q' \in \mathcal{P}$,*

i) $p \stackrel{*}{\Longrightarrow} p' \Rightarrow (\exists\, q' : q' \in \mathcal{P} : q \stackrel{*}{\Longrightarrow} q' \wedge p'\mathcal{R}q')$ *and*

ii) $q \stackrel{*}{\Longrightarrow} q' \Rightarrow (\exists\, p' : p' \in \mathcal{P} : p \stackrel{*}{\Longrightarrow} p' \wedge p'\mathcal{R}q')$.

Proof. It is straightforward to prove the two properties by induction on the number of actions needed to reach p' from p and q' from q, respectively. The proof is very similar to the proof of Lemma 2.2.21 on Page 13. □

Given Lemma 4.3.17 and two branching bisimilar P/T nets, it is not difficult to prove that any non-dead transition in one net corresponds to a non-dead transition with the same label in the other net. This leads easily to the desired congruence property.

Property 4.3.18. *For any two P/T nets (N_0, s_0) and (N_1, s_1) in $\mathcal{O}$,*
$(N_0, s_0) \sim_b (N_1, s_1) \Rightarrow \alpha(N_0, s_0) = \alpha(N_1, s_1)$.

Proof. Let $N_0 = (P_0, T_0, F_0, \ell_0)$ and $N_1 = (P_1, T_1, F_1, \ell_1)$. Assume that $(N_0, s_0) \sim_b (N_1, s_1)$. Let $\mathcal{R}$ be a branching bisimulation between (N_0, s_0) and (N_1, s_1), as defined in Definition 2.2.15 on Page 11. Recall that $\_ \Longrightarrow \_ \subseteq \mathcal{P} \times \mathcal{P}$, defined in Definition 2.2.13 also on Page 11, is the relation expressing reachability via silent actions.

Assume that $a \in \alpha(N_0, s_0)$. It follows from Definitions 4.3.16 (Alphabet) and 2.3.26 (Dead transition) on Page 24 that there must exist a reachable marking $u \in [N_0, s_0\rangle$ and a transition $t_0 \in T_0$ such that $(N_0, u)[t_0\rangle$, $a \neq \tau$, and $\ell(t_0) = a$. Hence, Lemma 4.3.17 *i)* and Definition 2.3.17 (Reachable markings) on Page 21 yield that there must exist a reachable marking $v \in [N_1, s_1\rangle$ such that $(N_0, u)\mathcal{R}(N_1, v)$. Since $\mathcal{R}$ is a branching bisimulation and $a \neq \tau$, there exist a marking w of N_1 such that $(N_1, v) \Longrightarrow (N_1, w)$ and a transition $t_1 \in T_1$ such that $\ell(t_1) = a$ and $(N_1, w)[t_1\rangle$. It follows that $w \in [N_1, s_1\rangle$. Hence, Definition 2.3.26 (Dead transition) implies that t_1 is not dead. Consequently, Definition 4.3.16 (Alphabet) yields that $a \in \alpha(N_1, s_1)$.

A symmetrical argument proves that any $a \in \alpha(N_1, s_1)$ is also an element of $\alpha(N_0, s_0)$, which completes the proof. □

Since an object life cycle does not have any dead transitions, it is straightforward to calculate its alphabet, as the following property shows.

Property 4.3.19. *For any object life cycle* $(N, [i])$ *in* $\mathcal{L}$ *with* $N = (P, T, F, \ell)$, $\alpha(N, [i]) = \{\ell(t) \mid t \in T \wedge \ell(t) \neq \tau\}$.

Proof. Definitions 4.3.2 (Object life cycle) on Page 161 and 4.3.16 (Alphabet) on the preceding page. □

The properties of the alphabet, encapsulation, and abstraction operators given in the previous section, namely Lemmas 4.2.8, 4.2.10, 4.2.11, 4.2.13, and 4.2.14, are easily translated to labeled P/T nets. They can be proven straightforwardly from Definitions 4.3.10 (Encapsulation) and 4.3.11 (Abstraction) on Page 165 and 4.3.16 (Alphabet) on the facing page.

Properties 4.2.12, 4.2.15, and 4.2.18 of Section 4.2.1 can now be proven within the framework of this section. The proofs use the basic lemmas for the alphabet, encapsulation, and abstraction operators mentioned above, as well as the congruence results of Properties 4.3.12 on Page 165 and 4.3.18 on the preceding page. They go along the same lines as the proofs in the previous section and are, hence, omitted.

The first property shows the existence of canonical sets of methods for proving relationships under protocol, projection, and protocol/projection inheritance.

Property 4.3.20. *For any object life cycles* $(N_0, [i]), (N_1, [i]) \in \mathcal{L}$,

i) $(N_1, [i]) \leq_{pt} (N_0, [i]) \Leftrightarrow \partial_{\alpha(N_1,[i]) \backslash \alpha(N_0,[i])}(N_1, [i]) \sim_b (N_0, [i])$,

ii) $(N_1, [i]) \leq_{pj} (N_0, [i]) \Leftrightarrow \tau_{\alpha(N_1,[i]) \backslash \alpha(N_0,[i])}(N_1, [i]) \sim_b (N_0, [i])$, *and*

iii) $(N_1, [i]) \leq_{pp} (N_0, [i]) \Leftrightarrow$
$\partial_{\alpha(N_1,[i]) \backslash \alpha(N_0,[i])}(N_1, [i]) \sim_b (N_0, [i]) \wedge \tau_{\alpha(N_1,[i]) \backslash \alpha(N_0,[i])}(N_1, [i]) \sim_b (N_0, [i])$.

As Property 4.2.12 on Page 151, Property 4.3.20 does not have a counterpart for life-cycle inheritance. However, to prove a life-cycle-inheritance relationship between two life cycles, it is always possible to choose a partitioning of exactly all the methods new in the subclass into two sets, one containing the methods that are encapsulated and the other one containing methods that are hidden.

The second property states that the inheritance relations are reflexive and transitive.

Property 4.3.21. *Protocol, projection, protocol/projection, and life-cycle inheritance, as defined in Definition 4.3.13 on Page 166, are preorders.*

The third property taken from the previous section shows that subclass equivalence under any form of inheritance corresponds to branching bisimilarity.

Definition 4.3.22. (Subclass equivalence) Let $\approx_*$, where $* \in \{pp, pt, pj, lc\}$, be the equivalence relation induced by $\leq_*$. For any object life cycles $(N_0, [i])$ and $(N_1, [i])$ in $\mathcal{L}$, $(N_0, [i]) \approx_* (N_1, [i]) \Leftrightarrow (N_0, [i]) \leq_* (N_1, [i]) \wedge (N_1, [i]) \leq_* (N_0, [i])$. The two life cycles are said to be *subclass equivalent* under $*$ inheritance.

Property 4.3.23. *For any object life cycles* $(N_0, [i]), (N_1, [i]) \in \mathcal{L}$ *and* $* \in \{pp, pt, pj, lc\}$, $(N_0, [i]) \approx_* (N_1, [i]) \Leftrightarrow (N_0, [i]) \sim_b (N_1, [i])$.

Another result of the previous section, namely Property 4.2.22 on Page 154, describes when a subclass can be refined to a more specialized subclass. It does not have a straightforward translation to the framework of this section. However, the transformation rules introduced in the next subsection can be used for this purpose.

A final aspect that needs to be studied is the decidability of the four inheritance relations.

Theorem 4.3.24. (Decidability of inheritance) *For any two object life cycles* $(N_0, [i])$ *and* $(N_1, [i])$ *in* $\mathcal{L}$*, it is decidable whether* $(N_1, [i])$ *is a subclass of* $(N_0, [i])$ *under any of the four inheritance relations of Definition 4.3.13 on Page 166.*

Proof. The first step in proving any inheritance relationship between two object life cycles is to choose appropriate sets of methods that must be encapsulated or hidden. For protocol, projection, and protocol/projection inheritance, Property 4.3.20 on the preceding page shows that canonical sets defined in terms of the alphabets of $(N_0, [i])$ and $(N_1, [i])$ can be chosen. Property 4.3.19 on the previous page implies that it is straightforward to calculate these two alphabets. For life-cycle inheritance, it is always possible to choose a partitioning of the methods in the alphabet of $(N_1, [i])$ and not in the alphabet of $(N_0, [i])$. Since the number of transitions of a P/T net is finite, there only exists a finite number of possibilities.

The second step involves checking branching bisimilarity. Observe that, according to Property 4.3.5 on Page 162, $(N_0, [i])$ and $(N_1, [i])$ are bounded. It follows from Definition 2.3.22 (Boundedness) on Page 23 and Definitions 4.3.10 (Encapsulation) and 4.3.11 (Abstraction) on Page 165 that any of the modified object life cycles used in the definition of the four inheritance relations, namely $\partial_H(N_1, [i])$, $\tau_I(N_1, [i])$, and $\tau_I \circ \partial_H(N_1, [i])$ with $H, I \subseteq E$, are also bounded. Obviously, the process corresponding to a bounded ordinary P/T net, defined by the operational semantics of Definition 4.3.1 on Page 160 and Definition 2.2.3 (Process) on Page 6, has a finite number of states and transitions. Hence, the processes corresponding to any of the life cycles or modified life cycles mentioned above are all finite. Consequently, checking any of the branching-bisimilarity relations occurring in Definition 4.3.13 on Page 166 is decidable. As a result, it is also decidable whether $(N_1, [i])$ is a subclass of $(N_0, [i])$ under any of the four inheritance relations. $\Box$

The key in the above decidability result is that all the P/T nets in the problem are bounded and, hence, all processes finite. As a result, branching bisimilarity and thus all the inheritance relations are decidable. The exact complexity of deciding branching bisimilarity on bounded ordinary P/T nets is unknown. It is known that deciding whether two finite processes are branching bisimilar can be done in polynomial time, where the size of the problem is defined as the number of states and transitions of the two processes (GROOTE AND VAANDRAGER, 1990). However, constructing the process corresponding to an ordinary P/T net requires in the worst case at least exponential space, where the size of the problem is the number of nodes of the P/T net. This space requirement is an immediate consequence of the known lowerbound on the complexity of the reachability problem (see ESPARZA AND NIELSEN, 1994). Hence, deciding any of the inheritance relations on two object life cycles cannot be done efficiently. This is one of the reasons to study in the next subsection transformation rules to construct subclasses from a given life cycle.

4.3.3 Transformation rules

An important goal of object-oriented design is to stimulate the *reuse* of software components. One of the aims of this chapter is to develop support for the reuse of object life cycles. Therefore, this subsection proposes a number of transformation rules that can be used to construct subclasses from a given object life cycle under specific forms of inheritance. Such transformation rules are useful for several other reasons as well. As long as object life cycles are not too complex, it is straightforward to verify whether there exists a specific inheritance relationship between them, both from a computational point of view and from a design point of view. However, the discussion at the end of the previous subsection shows that computation time might become unacceptable when the object life cycles become too large. In addition, even if there exists an inheritance relationship between two object life cycles, it might not always be meaningful from a design point of view. Therefore, a transformation rule for the construction of subclasses of object life cycles should preferably satisfy two criteria. First, it should be efficient in computation time. Second, it should represent a meaningful design construct. Unfortunately, these two criteria are often in conflict. Usually, the more general a transformation rule is, the more useful it is in practical design. The price to be paid is almost always the computation time needed to verify whether a specific transformation satisfies the requirements of the rule. A transformation rule that satisfies both the above mentioned criteria can be a useful aid in object-oriented design by stimulating the reuse of object life cycles.

This subsection presents four different transformation rules. Each one of them corresponds to a design construct which is often used in practice, namely choice, parallel composition, sequential composition, and iteration. The rules are inspired by the axioms of inheritance presented in Section 4.2.2. Each rule is a compromise between the two criteria mentioned above.

The following auxiliary definition is useful in the definition of the transformation rules.

Definition 4.3.25. (Union of labeled, ordinary P/T nets) Let $N_0 = (P_0, T_0, F_0, \ell_0)$ and $N_1 = (P_1, T_1, F_1, \ell_1)$ be two M-labeled, ordinary P/T nets such that $(P_0 \cup P_1) \cap (T_0 \cup T_1) = \emptyset$ and such that, for all $t \in T_0 \cap T_1$, $\ell_0(t) = \ell_1(t)$. The union $N_0 \cup N_1$ of N_0 and N_1 is the labeled, ordinary P/T net $(P_0 \cup P_1, T_0 \cup T_1, F_0 \cup F_1, \ell_0 \cup \ell_1)$. If two P/T nets satisfy the abovementioned two conditions, their union is said to be *well defined*.

The first transformation rule of this subsection preserves protocol inheritance. It is illustrated in Figure 4.3.26 on Page 173. The transformation rule is inspired by Axiom *PT* which is the algebraic axiom of inheritance defined in Property 4.2.24 on Page 156. Let N_0 and N be two labeled, ordinary P/T nets such that the union $N_1 = N_0 \cup N$ is well defined and such that $(N_0, [i])$ and $(N_1, [i])$ are object life cycles. Life cycle $(N_1, [i])$ is a subclass of $(N_0, [i])$ under protocol inheritance if the following two conditions are satisfied: *i)* N_0 and N do not share any transitions and *ii)* all transitions of N with input places in N_0 have a *visible* label which does not appear in the alphabet of $(N_0, [i])$. Transformation rule *PT* corresponds to the choice construct. Life cycle $(N_0, [i])$ is extended with one or more alternative branches of behavior defined by extension N. The second condition means that transitions of N with input places in N_0 act as guards, as explained in Section 4.2.2. It is not difficult to see that encapsulating the guards leads to a net whose behavior is identical to the behavior of the original life cycle.

Transformation rule *PT* is very general. Only two real restrictions are imposed on the extension N. The first one concerns the methods appearing in N. It is formulated in terms of the alphabet of $(N_0, [i])$. It follows from Property 4.3.19 on Page 169 that this restriction can be verified efficiently. The second restriction says that the resulting P/T net should again be a life cycle. Unfortunately, as explained in Section 4.3.1, this restriction can be hard to verify. However, Section 4.3.1 also contains some suggestions to overcome this problem. Therefore, the choice is made not to compromise the generality of the rule. Practical experience is needed to test the applicability of the rule.

Before formalizing rule *PT*, two remarks concerning notation are needed. First, for any *ordinary* P/T net, the bag of input nodes of a given node x, as introduced in Section 2.3.2, is simply the bag corresponding to the preset of x: $\boldsymbol{i}x = [y \mid y \in {}^\bullet x]$. Similarly, the bag of output nodes of x is the bag corresponding to the postset of x. This means that the bags of input and output nodes of x can be interpreted as sets. Therefore, in the remainder of this subsection, the preset and postset notation is also used to denote the bags of input and output nodes of a node, respectively. Second, in the definitions and the proofs of the transformation rules of this subsection, it is often convenient to be able to distinguish the preset and postset functions of the original object life cycle, the extension, and the resulting subclass. The standard notation does not allow such a distinction. In all the transformation rules, the net structures of the original life cycle, the extension, and the resulting subclass are denoted N_0, N, and N_1, respectively. Therefore, let ${}^\circ\_$, $\_^\circ$ denote the preset and postset functions of N_0; let ${}^\cdot\_$, $\_^\cdot$ be the preset and postset functions of N. The standard notation ${}^\bullet\_$, $\_^\bullet$ is used for the preset and postset functions of N_1. This notational convention carries over to the bags of input and output nodes of a given node.

Theorem 4.3.27. (Protocol-inheritance-preserving transformation rule *PT*) *Let* $N_0 = (P_0, T_0, F_0, \ell_0)$ *and* $N = (P, T, F, \ell)$ *be two labeled, ordinary P/T nets such that the union* $N_1 = N_0 \cup N$ *is well defined and such that* $(N_0, [i])$ *and* $(N_1, [i])$ *are object life cycles in* $\mathcal{L}$. *If two additional requirements are satisfied, namely*

i) $T_0 \cap T = \emptyset$ *and*

ii) $(\forall t : t \in T \wedge {}^\bullet t \cap P_0 \neq \emptyset : \ell(t) \in E \backslash \alpha(N_0, [i]))$,

then $(N_1, [i]) \leq_{pt} (N_0, [i])$.

Proof. According to Definition 4.3.13 *i)* (Protocol inheritance) on Page 166, it must be shown that there exists an $H \subseteq E$ such that $\partial_H(N_1, [i]) \sim_b (N_0, [i])$. Property 4.3.20 on

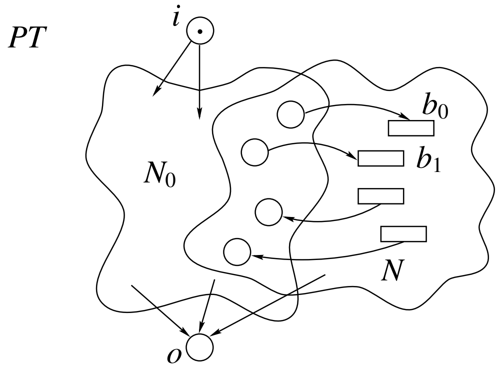


Figure 4.3.26: A protocol-inheritance-preserving transformation rule.

Page 169 suggests that H can be chosen equal to $\alpha(N_1, [i]) \backslash \alpha(N_0, [i])$. Let $\mathcal{R}$ be the relation $\{(\partial_H(N_1, u), (N_0, u)) \mid u \in \mathcal{B}(P_0)\}$. It is straightforward to prove that this relation is a branching bisimulation between $\partial_H(N_1, [i])$ and $(N_0, [i])$, as defined in Definition 2.2.15 on Page 11. Clearly, $\partial_H(N_1, [i])\mathcal{R}(N_0, [i])$. Assume that u is a marking in $\mathcal{B}(P_0)$, which implies that $\partial_H(N_1, u)\mathcal{R}(N_0, u)$. Three requirements have to be verified.

i) First, assume that $t \in T_0 \cup T$ is a transition of N_1 such that $(\partial_H(N_1, u)[t\rangle$ and such that $\partial_H(N_1, u)\ [\alpha\rangle\ \partial_H(N_1, u')$ where α is equal to the method identifier $(\ell_0 \cup \ell)(t)$ and where marking $u' \in \mathcal{B}(P_0 \cup P)$ is equal to $u - {}^\bullet t \uplus t^\bullet$. Requirement *i*) of Definition 2.2.15 implies that it must be shown that there exist two markings $v', v'' \in \mathcal{B}(P_0)$ such that $(N_0, u) \Longrightarrow (N_0, v'')\ [(\alpha)\rangle\ (N_0, v')$, $\partial_H(N_1, u)\mathcal{R}(N_0, v'')$, and $\partial_H(N_1, u')\mathcal{R}(N_0, v')$. It follows from the fact that u is a marking in $\mathcal{B}(P_0)$, Requirement *ii*) in the assumption of Theorem 4.3.27, and Definition 4.3.10 (Encapsulation) on Page 165, that t is a transition in T_0 and that u' is a marking in $\mathcal{B}(P_0)$. Consequently, $(N_0, u)\,[\ell_0(t)\rangle\,(N_0, u')$ with $\ell_0(t) = \alpha$. Hence, assuming that $v'' = u$ and $v' = u'$, the proof requirement is satisfied.

ii) Second, assume that $t \in T_0$ is a transition of N_0 such that $(N_0, u)[t\rangle$ and such that $(N_0, u)\,[\ell_0(t)\rangle\,(N_0, u')$ where $u' \in \mathcal{B}(P_0)$ is equal to $u - {}^\circ t \uplus t^\circ$. Clearly, this means that $\partial_H(N_1, u)\ [\ell_0(t)\rangle\ \partial_H(N_1, u')$. Thus, also Requirement *ii*) of Definition 2.2.15 is satisfied.

iii) Third, Requirement *iii*) of Definition 2.2.15 states that the following two implications must be shown: $\downarrow \partial_H(N_1, u) \Rightarrow \Downarrow (N_0, u)$ and $\downarrow (N_0, u) \Rightarrow \Downarrow \partial_H(N_1, u)$. Assume $\downarrow \partial_H(N_1, u)$. It follows from Definition 4.3.1 (Operational semantics) on Page 160 that $u = [o]$. Hence, also $\downarrow (N_0, u)$. Definition 2.2.14 ($\Downarrow$) on Page 11 yields immediately that $\Downarrow (N_0, u)$. The second implication is proven in the same way, which completes the proof.

□

Example 4.3.28. Consider Example 4.3.15 on Page 166. The protocol-inheritance relationships between life cycles $(N_1, [i])$ and $(N_0, [i])$ and between $(N_2, [i])$ and $(N_0, [i])$ can be easily proven by means of transformation rule *PT*. In both cases, method *repp* acts as the guard.

The remaining three transformation rules of this subsection are all based on the same principles. The rule that is the easiest one to understand is presented first. It preserves both protocol and projection inheritance. The rule is inspired by Axiom *PP*, as defined in Property 4.2.29 on Page 157. It is illustrated in Figure 4.3.29 on the next page. Let $(N_0, [i])$ be an object life cycle. Let N be a *connected, free-choice* P/T net such that the union $N_1 = N_0 \cup N$ is well defined. The marked P/T net $(N_1, [i])$ is a subclass of life cycle $(N_0, [i])$ under protocol/projection inheritance if the following four conditions are satisfied: *i*) N_0 and N only share a single place p; *ii*) all transitions of N have a label which does not appear in the alphabet of $(N_0, [i])$; *iii*) each transition of N with p as one of its input places has a *visible* label and *iv*) $(N, [p])$ is live and bounded. Transformation rule *PP* shows that under protocol/projection inheritance, it is allowed to *postpone* behavior. When $(N_1, [i])$ reaches a state in which place p is marked, it is possible to iterate the behavior defined by N an arbitrary number of times before continuing with the original behavior. The requirement that $(N, [p])$ is free-choice, live, and bounded guarantees that every token consumed from place p by a transition of N can always be returned to p. This property of N is crucial for the correctness of rule *PP*.

Transformation rule *PP* is fairly general. The only requirement that might seem out of place is the requirement that N is free-choice. However, as already explained in Section 2.3.5, free-choice P/T nets exhibit many interesting properties that do not carry over to general ordinary P/T nets. Two examples of such properties are monotonicity of liveness and monotonicity of home markings. These and other properties form the basis of transformation rule *PP*. At the end of this subsection, the role of the free-choice requirement in rule *PP* is discussed in more detail.

A positive consequence of the free-choice requirement is that it can be verified very efficiently whether a specific transformation satisfies the conditions of the transformation rule. Note that, in contrast to the previous transformation rule, rule *PP* does not require that $(N_1, [i])$ is an object life cycle. The fact that $(N_1, [i])$ is a life cycle follows from the other requirements. As already explained, the alphabet of an object life cycle is simply the set of its visible transition labels. It has also been mentioned that liveness and boundedness of free-choice P/T nets can be verified in linear time (KOVALYOV, 1996). This means that all requirements of rule *PP* can be verified efficiently.

The correctness of transformation rule *PP* is proven by showing that it is both a special case of transformation rule *PT* and of transformation rule *PJ* given further on in this subsection. The advantage of such a proof is that it is relatively short. The disadvantage is that it does not provide much insight in the exact working of the rule. To understand the details of rule *PP*, it is best to study the lemmas which form the basis of the proof of the two remaining transformation rules. These lemmas are given after the proof of transformation rule *PP*.

Finally, observe that, when Requirement *iii*) of rule *PP* is dropped, it does no longer preserve protocol inheritance, but that it still preserves projection inheritance.

Theorem 4.3.30. (Protocol/projection-inheritance-preserving transformation rule *PP*) *Let* $(N_0, [i])$ *with* $N_0 = (P_0, T_0, F_0, \ell_0)$ *be an object life cycle in* $\mathcal{L}$. *If* $N = (P, T, F,$

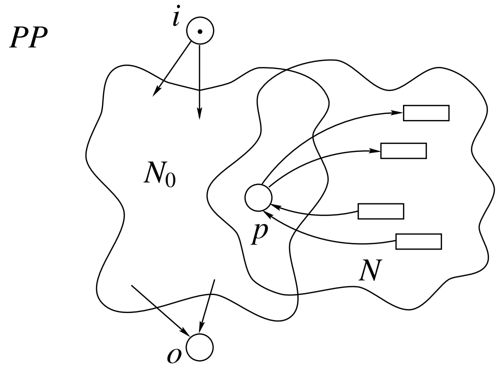


Figure 4.3.29: A protocol/projection-inheritance-preserving transformation rule.

ℓ) is a connected, labeled, free-choice P/T net with place $p \in P$ such that

i) $p \notin \{i, o\}$, $P_0 \cap P = \{p\}$, $T_0 \cap T = \emptyset$,

ii) $(\forall t : t \in T : \ell(t) \notin \alpha(N_0, [i]))$,

iii) $(\forall t : t \in T \wedge p \in {}^\bullet t : \ell(t) \neq \tau)$,

iv) $(N, [p])$ *is live and bounded, and*

v) $N_1 = N_0 \cup N$ *is well defined,*

then $(N_1, [i])$ *is an object life cycle in* $\mathcal{L}$ *such that* $(N_1, [i]) \leq_{pp} (N_0, [i])$.

Proof. To prove Theorem 4.3.30, it is shown that it is a special case of both Theorem 4.3.27 on Page 172 and Theorem 4.3.40 on Page 182. Thus, it is shown that the transformation rule preserves both protocol and projection inheritance, which according to Definition 4.3.13 (Inheritance relations) on Page 166 means that it also preserves protocol/projection inheritance. Let N_0, N, and N_1 be the three P/T nets satisfying the requirements of Theorem 4.3.30.

It is straightforward to verify that N_0, N, and N_1 satisfy the requirements of Theorem 4.3.27.

The second part of the proof starts by showing that N_0 contains a transition with place p in its postset. Since $(N_0, [i])$ is a life cycle, it follows from Definition 4.3.2 (Object life cycle) on Page 161 that N_0 is connected. It follows from Definition 2.3.15 (Connectedness) on Page 21 and the fact that places i and o are distinct that T_0 is not empty. Since a life cycle does not contain any dead transitions, Lemma 4.3.32 on the next page yields that there exists a reachable marking $s \in [N_0, [i]\rangle$ such that $s \geq [p]$. Definitions 2.3.17 (Reachable markings) on Page 21 and 2.3.7 (Firing rule) on Page 18 imply that there exists a transition $t_p \in T_0$ such that $p \in t_p{}^\circ$.

Transition t_p is used to construct a P/T net $N_2 = (P_2, T_2, F_2, \ell_2)$ as follows: $P_2 = P_0$, $T_2 = T \cup \{t_p\}$, $F_2 = F_0 \cup \{(p, t_p), (t_p, p)\}$, and $\ell_2 = \ell_0 \cup \{(t_p, \ell_0(t_p))\}$. It remains to be shown that N_0, N_1, and N_2 satisfy the requirements of Theorem 4.3.40 on Page 182, where N_2 place the role of N.

First, it must be shown that N_2 is connected and free-choice. It follows immediately from Definition 2.3.15 (Connectedness) on Page 21, the construction of N_2, and the fact that N is connected that also N_2 is connected. To prove that N_2 is free-choice, assume that N_2 is not free-choice. It follows from Definition 2.3.43 (Free-choice P/T net) on Page 27, the construction of N_2, and the fact that N is free-choice that there must be a transition $t \in T$ such that $\{p\} \subset {}^{\bullet}t$. Since $(N, [p])$ is live, there must be a reachable marking $s \in [N, [p]\rangle$ such that $(N, s)[t\rangle$. That is, $s > [p]$. However, by Property 2.3.23 (Characterization of boundedness) on Page 23, this contradicts the boundedness of $(N, [p])$. It follows that N_2 is free-choice.

Second, it is straightforward to see that N_0 and N_2 satisfy Requirements *i*) and *ii*) of Theorem 4.3.40.

Third, it must be shown that $(N_2, [p])$ is live and bounded. It follows from Definitions 2.3.33 (Trap) on Page 25 and 2.3.36 (Siphon) on Page 26 that the traps and siphons of N_2 are identical to the traps and siphons of N. It may be assumed that $T \neq \emptyset$. Otherwise, N_0 and N_1 are identical, which means that Theorem 4.3.30 is trivially true. It follows from the facts that both N and N_2 are connected and that $(N, [p])$ is live and Theorem 2.3.51 (Commoner's theorem) on Page 28 that also $(N_2, [p])$ is live. To prove that $(N_2, [p])$ is bounded, observe that the sets of reachable markings of $(N, [p])$ and $(N_2, [p])$ are identical. Thus, since $(N, [p])$ is bounded, also $(N_2, [p])$ is bounded.

Finally, it is straightforward to verify that $N_1 = (P_0, T_0, F_0 \backslash \{(t_p, p)\}, \ell_0) \cup (P_2, T_2, F_2 \backslash \{(p, t_p)\}, \ell_2)$. Thus, also Requirement *iv*) of Theorem 4.3.40 is satisfied, completing the proof. □

Example 4.3.31. Consider again the object life cycles in Example 4.3.15 on Page 166. It is not difficult to verify that the inheritance relationship between $(N_4, [i])$ and $(N_3, [i])$ can be proven by means of transformation rule *PP*.

Before continuing with the remaining two transformation rules of this subsection, a number of lemmas is introduced. These lemmas are needed in the correctness proofs of the rules. Most of the lemmas formulate properties that are useful to increase the understanding of the rules. If applicable, the lemmas are explained by means of transformation rule *PP* as defined in Theorem 4.3.30 on Page 174.

The first lemma is a simple auxiliary result. It shows that, in a connected, ordinary P/T net without dead transitions, it is possible to put a token in an arbitrary place.

Lemma 4.3.32. *Let $N = (P, T, F, \ell)$ be a connected, labeled, ordinary P/T net such that T is not empty. Assume that $s \in \mathcal{B}(P)$ is a marking such that (N, s) contains no dead transitions. For any place $p \in P$, there exists a reachable marking $s' \in [N, s\rangle$ such that $s' \geq [p]$.*

Proof. If N contains no places, then the lemma is trivial. Therefore, assume that $P \neq \emptyset$. Let p be a place in P. It follows from the assumption that T is not empty and Definition 2.3.15 (Connectedness) on Page 21 that there exists a transition $t \in {}^{\bullet}p \cup p^{\bullet}$. Since (N, s) has no dead transitions, Definition 2.3.26 (Dead transition) on Page 24 implies that there exists a marking $s' \in [N, s\rangle$ such that $s'[t\rangle$. If $t \in p^{\bullet}$, then $s' \geq [p]$. If $t \in {}^{\bullet}p$, Definition 2.3.7 (Firing rule) on Page 18 implies that $s'' = s' - {}^{\bullet}t \uplus t^{\bullet}$ is a marking in $[N, s\rangle$ such that $s'' \geq [p]$. □

As explained in Section 2.3.3, in the analysis of iterative behavior, home markings play an important role. Several results about home markings of live and bounded free-choice P/T nets are essential to the correctness of the transformation rules in this subsection. The following lemma is a technical result which characterizes the role of the empty marking in the analysis of home markings of live and bounded free-choice P/T nets.

Lemma 4.3.33. *Let $N = (P, T, F, \ell)$ be a connected, labeled, free-choice P/T net. Assume that $s \in \mathcal{B}(P)$ is a marking such that (N, s) is live and bounded. Let $s', s'' \in \mathcal{B}(P)$ be two markings of N.*

i) $\mathbf{0}$ is a home marking of $(N, \mathbf{0})$;

ii) if s' is a home marking of (N, s''), then $s' = \mathbf{0}$ if and only if $s'' = \mathbf{0}$.

Proof. The first part of the lemma is proven as follows. Since (N, s) is bounded, it follows from Property 2.3.63 on Page 32 that also $(N, \mathbf{0})$ is bounded. Hence, Property 2.3.23 (Characterization of boundedness) implies that the empty marking is the only reachable marking of $(N, \mathbf{0})$. Thus, it follows trivially from Definition 2.3.31 (Home marking) on Page 25 that $\mathbf{0}$ is a home marking of $(N, \mathbf{0})$.

The second part of the lemma follows directly from the following two observations. First, since s' is a home marking of (N, s''), it follows from Property 2.3.56 on Page 30 and Definition 2.3.31 (Home marking) on Page 25 that s' and s'' mark every S-component of N with the same number of tokens. Second, it follows from the fact that (N, s) is free-choice, live, and bounded and Theorem 2.3.61 (S-coverability) on Page 31 that every place of N is contained in the set of places of at least one S-component. □

The remaining lemmas all give results about a very specific kind of live and bounded free-choice P/T net, namely a free-choice P/T net N with a place p such that $(N, [p])$ is live and bounded. The extension N in transformation rule *PP* is such a P/T net. As mentioned earlier, the crucial property of N is that any token consumed from place p can always be returned. This property must remain valid under all circumstances. The following lemma states that a single token in place p can always be returned. That is, if life cycle $(N_0, [i])$ in rule *PP* puts a single token in place p, it is always possible to return this token when it is consumed by a transition of N.

Lemma 4.3.34. *Let $N = (P, T, F, \ell)$ be a connected, labeled, free-choice P/T net. If $p \in P$ is a place such that $(N, [p])$ is live and bounded, then $[p]$ is a home marking of $(N, [p])$.*

Proof. If N contains no transitions, the property is satisfied trivially. Therefore, assume that T is not empty. Definitions 2.3.26 (Dead transition) on Page 24 and 2.3.27 (Liveness) on Page 24 imply that, for any reachable marking $s \in [N, [p]\rangle$ and any transition $t \in T$, t is not dead in (N, s). Lemma 4.3.32 on the preceding page yields that, for any reachable marking $s \in [N, [p]\rangle$, there exists a marking $s' \in [N, s\rangle$ such that $s' \geq [p]$. Since $(N, [p])$ is bounded, it follows from Property 2.3.23 (Characterization of boundedness) on Page 23 that $s' = [p]$. Hence, Definition 2.3.31 (Home marking) on Page 25 yields that $[p]$ is a home marking of $(N, [p])$. □

The following corollary generalizes Lemma 4.3.34 to an arbitrary number of tokens in place p. Considering transformation rule *PP*, this result is needed because the original life cycle $(N_0, [i])$ may put multiple tokens in place p.

Corollary 4.3.35. *Let $N = (P, T, F, \ell)$ be a connected, labeled, free-choice P/T net. If $p \in P$ is a place such that $(N, [p])$ is live and bounded, then, for all positive natural numbers $n \in \mathbb{P}$, $[p^n]$ is a home marking of $(N, [p^n])$.*

Proof. Property 2.3.66 (Monotonicity of home markings) on Page 32 and Lemma 4.3.34 on the preceding page. □

Lemma 4.3.34 on the previous page and Corollary 4.3.35 show that any number of tokens put into place p by life cycle $(N_0, [i])$ in transformation rule *PP* can be returned when consumed by N, provided that no external effects disturb the process of returning tokens to p. Property 2.3.66 (Monotonicity of home markings) on Page 32 shows that this process is not disturbed when firing a transition of N_0 adds a token to p. This raises the question what happens if a transition of N_0 *removes* a token from p. Lemma 4.3.37 given below shows that the removal of token from p does not influence the process either. The following lemma is an auxiliary result needed in the proof of Lemma 4.3.37.

Lemma 4.3.36. *Let $N = (P, T, F, \ell)$ be a connected, labeled, free-choice P/T net. Assume that $p \in P$ is a place such that $(N, [p])$ is live and bounded. Let $s \in \mathcal{B}(P)$ be an arbitrary marking of N and $n \in \mathbb{P}$ a positive natural number. If (N, s) is live and, for every S-component (P_0, T_0, F_0) of N, $s(P_0) = n$, then $[p^n] \in [N, s\rangle$. That is, if (N, s) is live and s marks every S-component of N with n tokens, then marking $[p^n]$ is reachable from s.*

Proof. If N contains no transitions, it easily follows from the connectedness requirement that P contains only place p. As a result, the lemma is trivially satisfied. Therefore, assume that T is not empty. The proof is by induction on n.

Base case: Assume that $n = 1$. That is, it must be shown that $[p] \in [N, s\rangle$ from the assumption that s is a marking such that (N, s) is live and every S-component of N is marked with exactly one token.

It follows from Property 2.3.28 on Page 24 and Lemma 4.3.32 on Page 176 that there exists a marking $s' \in [N, s\rangle$ such that $s' \geq [p]$. It follows from the assumption that s marks every S-component of N with one token and Property 2.3.56 on Page 30 that also s' marks every S-component of N with one token. It follows from the fact that $(N, [p])$ is live and Property 2.3.58 on Page 30 that $[p]$ marks every S-component of N with one token. Consequently, for every S-component (P_0, T_0, F_0) of N, place p is an element of P_0. Recall that Theorem 2.3.61 (S-coverability) on Page 31 implies that every place of N is contained in the set of places of at least one S-component. Since s' marks every S-component of N with exactly one token and since $s' \geq [p]$, it follows that $s' = [p]$, which completes the proof.

Inductive step: The induction hypothesis states that, for some natural number $n \geq 1$ and any marking $s \in \mathcal{B}(P)$ such that (N, s) is live and s marks every S-component of N

with n tokens, $[p^n] \in [N, s\rangle$. Assume that $s \in \mathcal{B}(P)$ is a marking such that (N, s) is live and such that s marks every S-component of N with $n+1$ tokens. It must be shown that $[p^{n+1}] \in [N, s\rangle$.

Property 2.3.28 on Page 24 and Lemma 4.3.32 on Page 176 show that there exists a marking $s' \in [N, s\rangle$ such that $s' \geq [p]$. It follows from the assumption and Property 2.3.56 on Page 30 that s' marks every S-component of N with $n+1$ tokens. In the base case, it has been shown that $[p]$ marks every S-component of N with exactly one token. Since $s' \geq [p]$, marking $s' - [p]$ marks every S-component of N with n tokens. It follows from the fact that $(N, [p])$ is connected, live, and bounded with $T \neq \emptyset$ and Property 2.3.59 on Page 31 that $(N, s' - [p])$ is live. The induction hypothesis yields that $[p^n] \in [N, s' - [p]\rangle$. Property 2.3.21 (Monotonicity of reachable markings) on Page 22 yields that $[p^{n+1}] \in [N, s'\rangle$. Since $s' \in [N, s\rangle$, it follows that $[p^{n+1}] \in [N, s\rangle$, which completes the proof □

Lemma 4.3.37. *Let $N = (P, T, F, \ell)$ be a connected, labeled, free-choice P/T net. Assume that $p \in P$ is a place such that $(N, [p])$ is live and bounded. Let $s \in \mathcal{B}(P)$ be an arbitrary marking of N and $n \in \mathbb{P}$ a positive natural number. If $[p^n]$ is a home marking of $(N, [p] \uplus s)$, then $[p^{n-1}]$ is a home marking of (N, s), where $[p^0]$ corresponds to the empty bag $\mathbf{0}$.*

Proof. If $T = \emptyset$, the lemma is trivially true. Therefore, assume $T \neq \emptyset$. Two cases must be distinguished.

i) First, let $n = 1$. Assume that $[p]$ is a home marking of $(N, [p] \uplus s)$. It must be shown that $\mathbf{0}$ is a home marking of (N, s). Lemma 4.3.33 on Page 177 implies that it is necessary and sufficient to prove that $s = \mathbf{0}$.
It follows from the assumption and Definition 2.3.31 (Home marking) on Page 25 that $[p] \in [N, [p] \uplus s\rangle$. Since Property 2.3.66 (Monotonicity of home markings) on Page 32 and Lemma 4.3.34 on Page 177 imply that $(N, [p] \uplus s)$ is a home marking of $(N, [p] \uplus s)$, it follows that $[p] \uplus s \in [N, [p]\rangle$. However, according to Property 2.3.23 (Characterization of boundedness) on Page 23, this contradicts the boundedness of $(N, [p])$ unless $s = \mathbf{0}$, which completes this part of the proof.

ii) Second, let $n > 1$. Assume that $[p^n]$ is a home marking of $(N, [p] \uplus s)$. It must be shown that $[p^{n-1}]$ is a home marking of (N, s). The aim is to use Theorem 2.3.64 (Home-marking theorem) on Page 32. This means that it must be shown that *a*) (N, s) is bounded, *b*) (N, s) is live, *c*) $[p^{n-1}] \in [N, s\rangle$, and *d*) $[p^{n-1}]$ marks every proper trap of N.
Requirement *a*) follows simply from the fact that $(N, [p])$ is live and bounded and Property 2.3.63 on Page 32.
Requirement *b*) is proven as follows. Since $(N, [p])$ is connected, live, and bounded with $T \neq \emptyset$, it follows from Property 2.3.59 on Page 31 that $[p]$ marks every S-component of N with one token. Consequently, $[p^n]$ marks every S-component of N with n tokens. It follows from the assumption that $[p^n]$ is a home marking of $(N, [p] \uplus s)$ and Property 2.3.56 on Page 30 that $[p] \uplus s$ marks every S-component

of N also with n tokens. Hence, s marks every S-component with $n - 1$ tokens. Since $n > 1$, Property 2.3.59 on Page 31 yields that (N, s) is live.

Requirement $c)$ follows from Requirement $b)$, the observation made in the proof of Requirement $b)$ that s marks every S-component of N with $n - 1$ tokens, and Lemma 4.3.36 on Page 178.

Requirement $d)$ is proven by means of Theorem 2.3.31 (Home-marking theorem) on Page 25 and Lemma 4.3.34 on Page 177. It follows from these two results and the fact that $(N, [p])$ is connected, live, and bounded with $T \neq \emptyset$ that $[p]$ marks every proper trap of N. Hence, since $n > 1$, $[p^{n-1}]$ also marks every proper trap of N.

Combining Requirements $a)$ through $d)$, the Home-marking theorem yields that $[p^{n-1}]$ is a home marking of (N, s).

□

Let us return to transformation rule *PP* one more time. The results given so far show that, under all circumstances, it is possible to return all tokens in places of N to place p. Even the removal of a token from place p does not disturb this property, as shown by Lemma 4.3.37 above. This result suggests that it must be possible to return tokens to p without consuming any tokens from p in the process. Lemma 4.3.38 shows that this is indeed possible.

Recall from Section 2.3.3 that a sequence of length n, for some natural number n, over some alphabet of identifiers A is a function from $\{0, \ldots, n-1\}$ to A. An element $a \in A$ is said to be an element of a sequence σ over A of length n if and only if $a = \sigma(i)$, for some $0 \leq i < n$. The concatenation of two sequences σ and σ' over A of length n and m, respectively, denoted $\sigma\sigma'$, is the sequence of length $n + m$ defined as follows: for $0 \leq i < n$, $\sigma\sigma'(i) = \sigma(i)$ and, for $n \leq i < n + m$, $\sigma\sigma'(i) = \sigma'(i - n)$.

Lemma 4.3.38. *Let $N = (P, T, F, \ell)$ be a connected, labeled, free-choice P/T net. Assume that $p \in P$ is a place such that $(N, [p])$ is live and bounded. Let $s \in \mathcal{B}(P)$ be an arbitrary marking of N and $n \in \mathbb{N}$ an arbitrary natural number. If $[p^n]$ is a home marking of (N, s), then there exists a firing sequence $\sigma \in T^*$ such that $(N, s)\ [\sigma\rangle\ (N, [p^n])$ and, for all $t \in p^\bullet$, $t \notin \sigma$.*

Proof. The lemma is trivial if N contains no transitions. Therefore, assume that T is not empty. The proof is by induction on n.

Base case: Assume that $n = 0$. It must be shown that there exists a firing sequence $\sigma \in T^*$ such that $(N, s)\ [\sigma\rangle\ (N, \mathbf{0})$ and, for all $t \in p^\bullet$, $t \notin \sigma$, under the assumption that $\mathbf{0}$ is a home marking of (N, s).

Lemma 4.3.33 on Page 177 yields that $s = \mathbf{0}$. Clearly, the empty sequence ε satisfies the requirements, completing the proof of the base case.

Inductive step: The induction hypothesis states that for some $n \geq 0$ and any marking $s \in \mathcal{B}(P)$ such that $[p^n]$ is a home marking of (N, s), there exists a firing sequence $\sigma \in T^*$ such that $(N, s)\ [\sigma\rangle\ (N, [p^n])$ and, for all $t \in p^\bullet$, $t \notin \sigma$. Assume that $s \in \mathcal{B}(P)$ is a marking of N such that $[p^{n+1}]$ is a home marking of (N, s). It must be shown that there exists a firing sequence $\sigma \in T^*$ such that $(N, s)\,[\sigma\rangle\,(N, [p^{n+1}])$ and, for all $t \in p^\bullet$, $t \notin \sigma$.

It follows from the fact that $(N, [p])$ is connected, live, and bounded with $T \neq \emptyset$ and Property 2.3.59 on Page 31 that $[p]$ marks every S-component of N with exactly one token. As a result, $[p^{n+1}]$ marks every S-component of N with $n + 1$ tokens. Property 2.3.56 on Page 30 and the assumption that $[p^{n+1}]$ is a home marking of (N, s) imply that also s marks every S-component with $n + 1$ tokens. Thus, Property 2.3.59 on Page 31 yields that (N, s) is live. Consequently, it follows from Properties 2.3.19 on Page 22 and 2.3.28 on Page 24 and Lemma 4.3.32 on Page 176 that there exist a reachable marking $s' \in [N, s\rangle$ and a firing sequence $\sigma \in T^*$ such that $(N, s)\ [\sigma\rangle\ (N, s')$ and $s' \geq [p]$. Clearly, σ can be chosen such that, for all $t \in p^\bullet$, $t \notin \sigma$. Since $s' \in [N, s\rangle$ and $[p^{n+1}]$ is a home marking of (N, s), it follows from Definition 2.3.31 (Home marking) on Page 25 that $[p^{n+1}]$ is also a home marking of (N, s'). Since $s' \geq [p]$, Lemma 4.3.37 on Page 179 yields that $[p^n]$ is a home marking of $(N, s' - [p])$. Thus, it follows from the induction hypothesis that there exists a firing sequence $\sigma' \in T^*$ such that $(N, s' - [p])\ [\sigma'\rangle\ (N, [p^n])$ and, for all $t \in p^\bullet$, $t \notin \sigma'$. Property 2.3.21 (Monotonicity of reachable markings) on Page 22 yields that $(N, s')\ [\sigma'\rangle\ (N, [p^{n+1}])$. Concatenating firing sequences σ and σ', yields the desired firing sequence. That is, $(N, s)\ [\sigma\sigma'\rangle\ (N, [p^{n+1}])$ such that, for all $t \in p^\bullet$, $t \notin \sigma\sigma'$, which completes the proof. $\square$

The next transformation rule of this subsection, *PJ*, builds upon the results given so far. It is inspired by the algebraic axioms of inheritance *PJ*1 and *PJ*2 given in Property 4.2.27 on Page 157. Theorem 4.3.40 given below formalizes transformation rule *PJ*. Figure 4.3.39 on the following page illustrates the rule. It shows that rule *PJ* corresponds to a sequential composition. New behavior may be inserted between sequential parts of a life cycle, yielding a subclass under projection inheritance. In contrast to the previous two transformation rules, the original life cycle is modified. Basically, transformation rule *PJ* says that it is allowed to replace an arc in the original life cycle by an entire P/T net. The original life cycle $(N_0, [i])$ contains a place p which has a transition t_p as one of its input transitions. The modification of $(N_0, [i])$ is based upon a free-choice P/T net N sharing place p and transition t_p with N_0. Place p is the only input place of t_p in N. The result of the transformation rule is the P/T net N_1 obtained by taking the union of N_0 and N after removing both the arc between t_p and p from N_0 and the arc between p and t_p from N. The requirement that $(N, [p])$ is live and bounded guarantees that N_1 always has the option to move every token that transition t_p would normally have put into place p to place p by only firing transitions of N. The requirement that all transitions of N other than t_p are labeled with method identifiers not appearing in the alphabet of $(N_0, [i])$ guarantees that hiding these methods does not influence the behavior of the original life cycle.

As already mentioned, transformation rule *PP* is a special case of rule *PJ*. Similar to rule *PP*, the only compromise with respect to generality of *PJ* is the requirement that N is free-choice. The requirements of transformation rule *PJ* can be verified with the same efficiency as the requirements of *PP*.

Recall the notation concerning pre- and postsets introduced on Page 172.

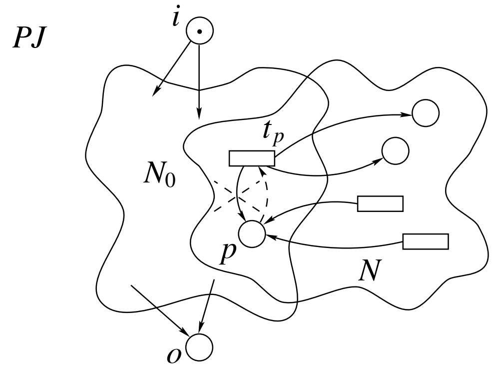


Figure 4.3.39: A projection-inheritance-preserving transformation rule.

Theorem 4.3.40. (Projection-inheritance-preserving transformation rule *PJ*) *Let* $(N_0, [i])$ *with* $N_0 = (P_0, T_0, F_0, \ell_0)$ *be an object life cycle in* $\mathcal{L}$. *If* $N = (P, T, F, \ell)$ *is a connected, labeled, free-choice P/T net with place* $p \in P$ *and transition* $t_p \in T$ *such that*

i) $p \notin \{i, o\}$, $P_0 \cap P = \{p\}$, $T_0 \cap T = \{t_p\}$, $(t_p, p) \in F_0$, *and* $^\bullet t_p = \{p\}$,

ii) $(\forall t : t \in T \backslash T_0 : \ell(t) \notin \alpha(N_0, [i]))$,

iii) $(N, [p])$ *is live and bounded, and*

iv) $N_1 = (P_0, T_0, F_0 \backslash \{(t_p, p)\}, \ell_0) \cup (P, T, F \backslash \{(p, t_p)\}, \ell)$ *is well defined,*

then $(N_1, [i])$ *is an object life cycle in* $\mathcal{L}$ *such that* $(N_1, [i]) \leq_{pj} (N_0, [i])$.

Proof. The proof consists of two parts. In the first part, a branching bisimulation is given showing that $(N_0, [i])$ and $(N_1, [i])$ satisfy the requirement of Definition 4.3.13 *ii*) (Projection inheritance) on Page 166. In the second part, the branching bisimulation is used to prove that $(N_1, [i])$ is an object life cycle.

In the first part of the proof, it must be shown that there exists an $I \subseteq E$, such that $\tau_I(N_1, [i]) \sim_b (N_0, [i])$. Let I be defined as the set $\alpha(N_1, [i]) \backslash \alpha(N_0, [i])$. Let $\mathcal{R}$ be the relation $\{(\tau_I(N_1, u), (N_0, v)) \mid u \in [N_1, [i]\rangle \wedge v \in [N_0, [i]\rangle \wedge u \lceil (P_0 \backslash \{p\}) = v \lceil (P_0 \backslash \{p\}) \wedge v \lceil \{p\}$ is a home marking of $(N, u \lceil P)\}$. The remainder of this part of the proof is devoted to showing that $\mathcal{R}$ is a branching bisimulation between $\tau_I(N_1, [i])$ and $(N_0, [i])$, as defined in Definition 2.2.15 on Page 11. Note that the abstraction operator τ_I does not affect the set of markings reachable from $(N_1, [i])$. Any firing sequence, as defined in Definition 2.3.18 on Page 22, enabled in $(N_1, [i])$ is also enabled in $\tau_I(N_1, [i])$. Property 2.3.19 on Page 22 implies that the sets of reachable markings of $(N_1, [i])$ and $\tau_I(N_1, [i])$ are identical.

First, it must be shown that $\tau_I(N_1, [i]) \mathcal{R} (N_0, [i])$. Clearly, it suffices to show that $\mathbf{0}$ is a home marking of $(N, \mathbf{0})$, which follows immediately from Lemma 4.3.33 on Page 177.

Second, it must be shown that relation $\mathcal{R}$ is a branching bisimulation. Assume that $u \in [N_1, [i]\rangle$ and $v \in [N_0, [i]\rangle$ are markings such that $\tau_I(N_1, u) \mathcal{R} (N_0, v)$, which implies that $u \lceil (P_0 \backslash \{p\}) = v \lceil (P_0 \backslash \{p\})$ and $v \lceil \{p\}$ is a home marking of $(N, u \lceil P)$.

i) Assume that $t \in T_0 \cup T$ is a transition of N_1 such that $\tau_I(N_1, u)[t\rangle$ and such that $\tau_I(N_1, u)\ [\alpha\rangle\ \tau_I(N_1, u')$ where α is equal to the method identifier $(\ell_0 \cup \ell)(t)$ and where marking $u' \in [N_1, [i]\rangle$ is equal to $u - {}^\bullet t \uplus t^\bullet$. Requirement *i*) of Definition 2.2.15 states that it must be shown that there exist two markings $v', v'' \in \mathcal{B}(P_0)$ of N_0 such that *a*) $(N_0, v) \Longrightarrow (N_0, v'')\ [(\alpha)\rangle\ (N_0, v')$, *b*) $\tau_I(N_1, u)\mathcal{R}(N_0, v'')$, and *c*) $\tau_I(N_1, u')\mathcal{R}(N_0, v')$. Two cases must be distinguished.

First, assume that $t \in T \backslash T_0$. That is, t is a transition of N different from t_p. It follows from Requirement *ii*) of Theorem 4.3.40 and Definition 4.3.11 (Abstraction) on Page 165 that $\alpha = \tau$. Let $v'' = v' = v$.

Requirement *a*) follows easily from the fact that $\alpha = \tau$.

Requirement *b*) follows immediately from the assumption that $\tau_I(N_1, u)\mathcal{R}(N_0, v)$.

To prove Requirement *c*), it must be shown that $\tau_I(N_1, u')\mathcal{R}(N_0, v)$. By assumption, $u' \in [N_1, [i]\rangle$ and $v \in [N_0, [i]\rangle$. Since $t \in T\backslash T_0$, Requirement *i*) of Theorem 4.3.40 implies that $u' \restriction (P_0\backslash\{p\}) = u \restriction (P_0\backslash\{p\})$. Since $u \restriction (P_0\backslash\{p\}) = v \restriction (P_0\backslash\{p\})$, it follows that $u' \restriction (P_0\backslash\{p\}) = v \restriction (P_0\backslash\{p\})$. It remains to be shown that $v \restriction \{p\}$ is a home marking of $(N, u' \restriction P)$. This result follows easily from Definition 2.3.31 (Home marking) on Page 25 and the observations that $\tau_I(N_1, u)[\alpha\rangle\tau_I(N_1, u')$ and that $v \restriction \{p\}$ is a home marking of $(N, u \restriction P)$.

Second, assume that $t \in T_0$. That is, t is a transition of N_0. Let $v'' = v$ and $v' = v - {}^\circ t \uplus t^\circ$.

To prove Requirement *a*), it is needed that $u(p) \leq v(p)$. That is, marking v marks place p with at least as many tokens as u. This property is proven as follows. Since $T \neq \emptyset$ and $(N, [p])$ is connected, free-choice, live, and bounded, Property 2.3.58 on Page 30 implies that $[p]$ marks every S-component of N. Clearly, this means that place p is contained in every S-component of N. Consequently, $v \restriction \{p\}$ marks every S-component of N with $v(p)$ tokens. Since $v \restriction \{p\}$ is a home marking of $(N, u \restriction P)$, Property 2.3.56 on Page 30 proves that $u(p) \leq v(p)$. Since $\tau_I(N_1, u)[t\rangle$, $\alpha = (\ell_0 \cup \ell)(t)$, $\tau_I(N_1, u)[\alpha\rangle\tau_I(N_1, u')$, $u \restriction (P_0\backslash\{p\}) = v \restriction (P_0\backslash\{p\})$, $u(p) \leq v(p)$, and $t \in T_0$, it follows that $(N_0, v)[t\rangle$ and $(N_0, v)\ [\alpha\rangle\ (N_0, v - {}^\circ t \uplus t^\circ)$. Requirement *a*) easily follows from the fact that $v'' = v$ and $v' = v$.

Requirement *b*) follows immediately from the assumptions that $\tau_I(N_1, u)\mathcal{R}(N_0, v)$ and that $v'' = v$.

Finally, Requirement *c*) is proven as follows. The assumptions and Requirement *a*) yield that $u' \in [N_1, [i]\rangle$ and $v' \in [N_0, [i]\rangle$. It follows from the fact that $u \restriction (P_0\backslash\{p\}) = v \restriction (P_0\backslash\{p\})$, and the definitions of u' and v' that $u' \restriction (P_0\backslash\{p\}) = v' \restriction (P_0\backslash\{p\})$. It remains to be shown that $v' \restriction \{p\}$ is a home marking of $(N, u' \restriction P)$. Recall that the markings u and v satisfy the property that $v \restriction \{p\}$ is a home marking of $(N, u \restriction P)$. Five cases can be distinguished. In this case analysis, Requirements *i*) and *iv*) of Theorem 4.3.40 are used to determine expressions for $v' \restriction \{p\}$ and $u' \restriction P$. Recall that $u' = u - {}^\circ t \uplus t^\circ$ and $v' = v - {}^\bullet t \uplus t^\bullet$. First, if $p \notin {}^\bullet t \cup t^\bullet$ or if $p \in {}^\bullet t \cap t^\bullet$ and $t \neq t_p$, then $v' \restriction \{p\} = v \restriction \{p\}$ and $u' \restriction P = u \restriction P$. Hence, the desired result simply follows from the above property. Second, assume that $p \in {}^\bullet t \cap t^\bullet$ and $t = t_p$. It follows that $p \in {}^\circ t \cap t^\circ$. Hence, $v' \restriction \{p\} = v \restriction \{p\}$. It follows from the assumption that $p \in {}^\bullet t$ and the fact that ${}^\cdot t_p = \{p\}$ that ${}^\bullet t \restriction P = \{p\}$. Since $p \in t^\bullet$, it follows that $p \in t^{\cdot}$. Since ${}^{\cdot}t_p = \{p\}$ and $(N, [p])$ is bounded, Property 2.3.23 (Characterization of boundedness) on Page 23 yields that $t_p{}^{\cdot} = \{p\}$. Consequently, $t^{\bullet} \restriction P = \{p\}$. Thus, $u' \restriction P = u \restriction P$. As in the previous case, the desired result simply follows from the abovementioned property. Third, assume that $p \in {}^{\bullet}t \backslash t^{\bullet}$ and $t \neq t_p$. It follows from $p \in {}^{\bullet}t$ that $p \in {}^{\circ}t$. It follows from $p \notin t^{\bullet}$ and $t \neq t_p$ that $p \notin t^{\circ}$. Hence, $v' \restriction \{p\} = v \restriction \{p\} - [p]$ and $u' \restriction P = u \restriction P - [p]$. As a result, the desired result follows from the above property and Lemma 4.3.37 on Page 179. Fourth, assume that $p \in {}^{\bullet}t \backslash t^{\bullet}$ and $t = t_p$. It follows from $p \in {}^{\bullet}t$ and ${}^{\cdot}t_p = \{p\}$ that ${}^{\bullet}t \restriction P = \{p\} = {}^{\cdot}t$. It follows from the requirements of Theorem 4.3.40 that $t_p{}^{\bullet} \restriction P = t_p{}^{\cdot}$. Hence, $u' \restriction P = u \restriction P - {}^{\cdot}t \uplus t^{\cdot}$. It follows from $p \in {}^{\bullet}t$ that $p \in {}^{\circ}t$. It follows from $t = t_p$ that $p \in t^{\circ}$. Hence, $v' \restriction \{p\} = v \restriction \{p\}$. Since, by assumption, $\tau_I(N_1, u)[t\rangle$ and $p \in {}^{\bullet}t$, it follows from the fact that ${}^{\cdot}t = \{p\}$ that also $(N, u \restriction P)[t\rangle$. Consequently, $u' \restriction P \in [N, u \restriction P\rangle$. Thus, Definition 2.3.31 (Home marking) on Page 25, the fact that $v' \restriction \{p\} = v \restriction \{p\}$, and the assumption that $v \restriction \{p\}$ is a home marking of $(N, u \restriction P)$ yield that $v' \restriction \{p\}$ is a home marking of $(N, u' \restriction P)$. Finally, if $p \in t^{\bullet} \backslash {}^{\bullet}t$, then also $p \in t^{\circ} \backslash {}^{\circ}t$. Thus, $v' \restriction \{p\} = v \restriction \{p\} \uplus [p]$ and $u' \restriction P = u \restriction P \uplus [p]$. In this case, the desired result follows from the above property and Property 2.3.66 (Monotonicity of home markings) on Page 32.

ii) Assume that $t \in T_0$ is a transition of N_0 such that $(N_0, v)[t\rangle$ and such that $(N_0, v)[\alpha\rangle (N_0, v')$ where α is equal to the method identifier $\ell_0(t)$ and where marking $v' \in [N_0, [i]\rangle$ is equal to $v - {}^{\circ}t \uplus t^{\circ}$. Requirement *ii*) of Definition 2.2.15 states that it must be shown that there exist markings $u', u'' \in \mathcal{B}(P_0 \cup P)$ such that *a*) $\tau_I(N_1, u) \Longrightarrow \tau_I(N_1, u'')\ [(\alpha)\rangle\ \tau_I(N_1, u')$, *b*) $\tau_I(N_1, u'')\mathcal{R}(N_0, v)$, and *c*) $\tau_I(N_1, u')\mathcal{R}(N_0, v')$.

Let $u'' = v$ and $u' = v - {}^{\bullet}t \uplus t^{\bullet}$. This choice for u'' and u' is inspired by the fact that it is possible to reach marking v from u by firing as many transitions of N as necessary. Since marking v enables transition t, it is straightforward to prove the desired requirements.

Requirement *a*) is proven as follows. The assumption that $v \restriction \{p\}$ is a home marking of $(N, u \restriction P)$, Requirements *i*) and *iii*) of Theorem 4.3.40, and Lemma 4.3.38 on Page 180 yield that there exists a firing sequence $\sigma \in (T \backslash \{t_p\})^*$ such that $(N, u \restriction P)\ [\sigma\rangle\ (N, v \restriction \{p\})$. The assumption that $u \restriction (P_0 \backslash \{p\}) = v \restriction (P_0 \backslash \{p\})$ and Requirement *iv*) of Theorem 4.3.40 imply that $\tau_I(N_1, u)\ [\sigma\rangle\ (N_1, v)$. Definition 4.3.11 (Abstraction) on Page 165, the definition of I, and Requirement *ii*) of Theorem 4.3.40 yield that $\tau_I(N_1, u) \Longrightarrow \tau_I(N_1, v)$. It easily follows from $(N_0, v)[t\rangle$, the definition of u', the fact that $\ell_0(t) = \alpha$, and Definition 2.3.7 (Firing rule) on Page 18 that $\tau_I(N_1, v)\ [\alpha\rangle\ \tau_I(N_1, u')$, which completes the proof of Requirement *a*).

To prove Requirement *b*), it must be shown that $\tau_I(N_1, v)\mathcal{R}(N_0, v)$. By assumption, $v \in [N_0, [i]\rangle$. Since, also by assumption, $u \in [N_1, [i]\rangle$, Requirement *a*) shows that $v \in [N_1, [i]\rangle$. Clearly, also $v \restriction (P_0 \backslash \{p\}) = v \restriction (P_0 \backslash \{p\})$. Thus, since $v \restriction P = v \restriction \{p\}$, Requirement *b*) follows from Corollary 4.3.35 on Page 178 and Lemma 4.3.33 *i*) on Page 177.

To prove Requirement *c*), two cases must be distinguished. First, assume that $t \neq$

t_p. It follows from the requirements of Theorem 4.3.40 that ${}^\bullet t = {}^\circ t$ and $t^\bullet = t^\circ$. Thus, Requirement *c*) simplifies to $\tau_I(N_1, v')\mathcal{R}(N_0, v')$, which is proven in a similar way as Requirement *b*). Second, assume that $t = t_p$. It follows from the assumptions and the proof of Requirement *a*) that $v' \in [N_0, [i]\rangle$ and $u' \in [N_1, [i]\rangle$. It follows from the requirements of Theorem 4.3.40 that ${}^\bullet t = {}^\circ t$ and $t^\bullet = t^\circ \backslash \{p\} \cup t^\cdot$. The definitions of v' and u' yield that $u' \lceil (P_0 \backslash \{p\}) = v' \lceil (P_0 \backslash \{p\})$. It remains to be shown that $v' \lceil \{p\}$ is a home marking of $(N, u' \lceil P)$. Again two cases need to be distinguished. If $p \in {}^\circ t$, Requirement *i*) of Theorem 4.3.40 yields that $v' \lceil \{p\} = v \lceil \{p\}$. Since $t = t_p$, ${}^\bullet t = {}^\circ t$, $t^\bullet = t^\circ \backslash \{p\} \cup t^\cdot$, and ${}^\cdot t_p = \{p\}$, it follows that $u' \lceil P = (v - {}^\bullet t \uplus t^\bullet) \lceil P = v \lceil \{p\} - [p] \uplus t^\circ = v \lceil \{p\} - {}^\cdot t \uplus t^\cdot$. Definition 2.3.7 (Firing rule) on Page 18 yields that $(N, v \lceil \{p\})\ [\ell(t)\rangle\ (N, u' \lceil P)$. Since $p \in {}^\circ t$ and $(N_0, v)[t\rangle$, it follows that $v \geq [p]$. Thus, Corollary 4.3.35 on Page 178 yields that $v \lceil \{p\}$ is a home marking of $(N, v \lceil \{p\})$. It follows from Definition 2.3.31 (Home marking) on Page 25 and the facts that $v' \lceil P = v \lceil P$ and that $(N, v \lceil \{p\})\ [\ell(t)\rangle\ (N, u' \lceil P)$ that $v' \lceil \{p\}$ is a home marking of $(N, u' \lceil P)$, which completes the proof in this case. If $p \notin {}^\circ t$, then $v' \lceil \{p\} = v \lceil \{p\} \uplus [p]$. Since ${}^\bullet t = {}^\circ t$ and $t^\bullet = t^\circ \backslash \{p\} \cup t^\cdot$, it follows that $u' \lceil P = v \lceil \{p\} \uplus t^\cdot$. Corollary 4.3.35 on Page 178 yields that $v \lceil \{p\} \uplus [p]$ is a home marking of $(N, v \lceil \{p\} \uplus [p])$. It follows from Definition 2.3.7 (Firing rule) that $(N, v \lceil \{p\} \uplus [p])\ [\ell(t)\rangle\ (N, u' \lceil P)$. Definition 2.3.31 (Home marking) yields that $v' \lceil \{p\}$ is a home marking of $(N, u' \lceil P)$, which completes the proof also in this case.

iii) Requirement *iii*) of Definition 2.2.15 (Branching bisimilarity) on Page 11 states that the following two implications must be shown: $\downarrow \tau_I(N_1, u) \Rightarrow \Downarrow (N_0, v)$ and $\downarrow (N_0, v) \Rightarrow \Downarrow \tau_I(N_1, u)$.

First, assume that $\downarrow \tau_I(N_1, u)$. It follows from Definition 4.3.1 (Operational semantics) on Page 160 that $u = [o]$. It follows from Requirement *i*) of Theorem 4.3.40 that $u \lceil (P_0 \backslash \{p\}) = [o]$ and $u \lceil P = \mathbf{0}$. Since $\tau_I(N_1, u)\mathcal{R}(N_0, v)$, it follows from the definition of $\mathcal{R}$ that $v \lceil (P_0 \backslash \{p\}) = [o]$ and that $v \lceil \{p\}$ is a home marking of $(N, \mathbf{0})$. Lemma 4.3.33 on Page 177 yields that $v \lceil \{p\} = \mathbf{0}$. Hence, $v = [o]$. Clearly, Definition 4.3.1 (Operational semantics) on Page 160 yields that $\downarrow (N_0, v)$. It follows from Definition 2.2.14 ($\Downarrow$) on Page 11 that $\Downarrow (N_0, v)$, which completes the proof of the first implication.

Second, assume that $\downarrow (N_0, v)$. It follows from Definition 4.3.1 (Operational semantics) on Page 160 that $v = [o]$. It follows from Requirement *i*) of Theorem 4.3.40 and the definition of $\mathcal{R}$ that $u \lceil (P_0 \backslash \{p\}) = [o]$ and that $\mathbf{0}$ is a home marking of $(N, u \lceil P)$. Lemma 4.3.33 on Page 177 yields that $u \lceil P = \mathbf{0}$. As a result, $u = [o]$. Hence, $\downarrow \tau_I(N_1, u)$, which implies that $\Downarrow \tau_I(N_1, u)$. This last result proves the second implication, completing the proof that $\mathcal{R}$ satisfies Requirement *iii*) of Definition 2.2.15.

At this point, it has been shown that relation $\mathcal{R}$ as defined above is a branching bisimulation. As a result, $(N_1, [i])$ is a subclass of $(N_0, [i])$ under projection inheritance, provided that $(N_1, [i])$ is a life cycle. The second part of the proof is devoted to this last requirement.

It is not difficult to verify that $(N_1, [i])$ satisfies Requirements *i*) through *iv*) of Definition 4.3.2 (Object life cycle) on Page 161. The remainder of the proof shows that $(N_1, [i])$ also satisfies Requirements *v*), *vi*), and *vii*). It uses the fact that the relation $\mathcal{R}$ of the first part of the proof is a branching bisimulation between $\tau_I(N_1, [i])$ and $(N_0, [i])$, with $I = \alpha(N_1, [i]) \backslash \alpha(N_0, [i])$.

Requirement *v*) (Proper termination) means that the following must be shown: For every reachable marking $s \in [N_1, [i]\rangle$, $o \in s$ implies $s = [o]$. Let $u \in [N_1, [i]\rangle$ be a reachable marking of $(N_1, [i])$ such that $o \in u$. Since $\mathcal{R}$ is a branching bisimulation between $\tau_I(N_1, [i])$ and $(N_0, [i])$, Lemma 4.3.17 on Page 168 implies that there exists a marking $v \in [N_0, [i]\rangle$ such that $\tau_I(N_1, u)\mathcal{R}(N_0, v)$. Consequently, $u \restriction (P_0\backslash\{p\}) = v \restriction (P_0\backslash\{p\})$ and $v \restriction \{p\}$ is a home marking of $(N, u \restriction P)$. Since $o \in u$ and $o \in P_0\backslash\{p\}$, this means that $o \in v$. It follows from the fact that $(N_0, [i])$ is an object life cycle and Requirement *v*) (Proper termination) of Definition 4.3.2 (Object life cycle) on Page 161 that $v = [o]$. This yields that $u \restriction (P_0\backslash\{p\}) = [o]$. Another consequence of the fact that $v = [o]$ is that $v \restriction \{p\} = \mathbf{0}$, which means that $\mathbf{0}$ is a home marking of $(N, u \restriction P)$. Lemma 4.3.33 on Page 177 yields that $u \restriction P = \mathbf{0}$. Since $u \restriction (P_0\backslash\{p\}) = [o]$, $u = u \restriction (P_0\backslash\{p\}) \uplus u \restriction P = [o] \uplus \mathbf{0} = [o]$, which proves that $(N_1, [i])$ satisfies Requirement *v*) (Proper termination) of Definition 4.3.2 on Page 161.

To prove that $(N_1, [i])$ satisfies Requirement *vi*) (Termination option) of Definition 4.3.2, it must be shown that, for every reachable marking $s \in [N_1, [i]\rangle$, $[o] \in [N_1, s\rangle$. Let $u \in [N_1, [i]\rangle$ be a reachable marking of $(N_1, [i])$. As before, using the fact that $\mathcal{R}$ is a branching bisimulation between $\tau_I(N_1, [i])$ and $(N_0, [i])$, Lemma 4.3.17 on Page 168 yields that there exists a marking $v \in [N_0, [i]\rangle$ such that $\tau_I(N_1, u)\mathcal{R}(N_0, v)$. Consequently, $u \restriction (P_0\backslash\{p\}) = v \restriction (P_0\backslash\{p\})$ and $v \restriction \{p\}$ is a home marking of $(N, u \restriction P)$. Since $(N, [p])$ is live and bounded, Lemma 4.3.38 on Page 180 and Requirement *i*) of Theorem 4.3.40 yield that there exists a firing sequence $\sigma \in (T\backslash\{t_p\})^*$ such that $(N, u \restriction P)\,[\sigma\rangle\,(N, v \restriction \{p\})$. It follows from Requirements *i*) and *iv*) of Theorem 4.3.40 that $(N_1, u)\,[\sigma\rangle\,(N_1, v)$. It remains to be shown that $[o]$ can be reached from marking v. It follows from the fact that $(N_0, [i])$ is an object life cycle, the fact that $v \in [N_0, [i]\rangle$, and Requirement *vi*) (Termination option) of Definition 4.3.2 (Object life cycle) on Page 161 that $[o] \in [N_0, v\rangle$. Let $\sigma_0 \in T_0^*$ be a firing sequence such that $(N_0, v)\,[\sigma_0\rangle\,(N_0, [o])$. It is possible to transform σ_0 into a firing sequence $\bar{\sigma_0} \in (T_0 \cup T)^*$ such that $(N_1, v)\,[\bar{\sigma_0}\rangle\,(N_1, [o])$. This transformation is inductively defined as follows. If $t_p \notin \sigma_0$, then $\bar{\sigma_0} = \sigma_0$. Clearly, this firing sequence satisfies the requirement. If $t_p \in \sigma_0$, then assume that σ_0 is of the form $\sigma_1 t_p \sigma_2$ with $t_p \notin \sigma_1$. Let $v' \in \mathcal{B}(P_0)$ be the marking of N_0 such that $(N_0, v)\,[\sigma_1 t_p\rangle\,(N_0, v')\,[\sigma_2\rangle\,(N_0, [o])$. It follows from the requirements of Theorem 4.3.40 that $(N_1, v)\,[\sigma_1 t_p\rangle\,(N_1, v' - [p] \uplus t_p{}^\bullet)$. Note that $v' \geq [p]$. Corollary 4.3.35 on Page 178 implies that $v' \restriction \{p\}$ is a home marking of $(N, v' \restriction \{p\})$. Since ${}^\bullet t_p = p$, it follows from Definition 2.3.7 (Firing rule) on Page 18 that $(N, v' \restriction \{p\})\,[\ell(t)\rangle\,(N, v' \restriction \{p\} - [p] \uplus t_p{}^\bullet)$. Definition 2.3.31 (Home marking) on Page 25 implies that $v' \restriction \{p\}$ is also a home marking of $(N, v' \restriction \{p\} - [p] \uplus t_p{}^\bullet)$. Lemma 4.3.38 on Page 180 yields that there exists a firing sequence $\sigma_3 \in (T\backslash\{t_p\})^*$ such that $(N, v' \restriction \{p\} - [p] \uplus t_p{}^\bullet)\,[\sigma_3\rangle\,(N, v' \restriction \{p\})$. Consequently, also $(N_1, v' - [p] \uplus t_p{}^\bullet)\,[\sigma_3\rangle\,(N_1, v')$. Thus, $(N_1, v)\,[\sigma_1 t_p \sigma_3\rangle\,(N_1, v')$. Choosing $\bar{\sigma_0} = \sigma_1 t_p \sigma_3 \bar{\sigma_2}$, it follows that $(N_1, v)\,[\bar{\sigma_0}\rangle\,(N_1, [o])$.

Since $(N_1, u)[\sigma\rangle(N_1, v)$, it follows that $(N_1, u)[\sigma\bar{\sigma}_0\rangle(N_1, [o])$, proving that $[o] \in [N_1, u\rangle$. Thus, $(N_1, [i])$ satisfies Requirement *vi)* (Termination option) of Definition 4.3.2.

It remains to prove Requirement *vii)* of Definition 4.3.2 saying that $(N_1, [i])$ contains no dead transitions, as defined in Definition 2.3.26 on Page 24. Since N_1 is defined as the union of N_0 and N, the set of transitions of N_1 is the set $T_0 \cup T$. The goal is to construct, for every transition of N_1, a firing sequence leading to a marking that enables that transition. First, let t be a transition in T_0. It follows from the fact that $(N_0, [i])$ is an object life cycle that $(N_0, [i])$ has no dead transitions. Consequently, there exist a firing sequence $\sigma \in T_0^*$ and a marking $s \in \mathcal{B}(P_0)$ such that $(N_0, [i])\ [\sigma\rangle\ (N_0, s)$ and $(N_0, s)[t\rangle$. Let $\bar{\sigma}$ be the firing sequence in $(T_0 \cup T)^*$ that is the result of transforming σ according to the transformation introduced in the previous part of the proof. It follows that $(N_1, [i])\ [\bar{\sigma}\rangle\ (N_1, s)$ and $(N_1, s)[t\rangle$. Thus, t is not dead in $(N_1, [i])$. Second, let t be a transition in $T\backslash\{t_p\}$. Since $(N_0, [i])$ does not contain any dead transitions, there exists a firing sequence $\sigma \in (T_0\backslash\{t_p\})^*$ and a marking $s \in \mathcal{B}(P_0)$ such that $(N_0, [i])\ [\sigma t_p\rangle\ (N_0, s)$. Note that $s \geq [p]$. It follows from the requirements of Theorem 4.3.40 that $(N_1, [i])\ [\sigma t_p\rangle\ (N_1, s - [p] \uplus t_p{}^\bullet)$. Since $(N, [p])$ is live, it follows from Definitions 2.3.27 (Liveness) on Page 24 and 2.3.7 (Firing rule) on Page 18 and the fact that ${}^\bullet t_p = \{p\}$ that also $(N, t_p{}^\bullet)$ is live. As a result of Property 2.3.28 on Page 24, transition t is not dead in $(N, t_p{}^\bullet)$. It follows that there is a firing sequence $\sigma_0 \in T^*$ and a marking $s' \in \mathcal{B}(P)$ such that $(N, t_p{}^\bullet)\ [\sigma_0\rangle\ (N, s')$ and $(N, s')[t\rangle$. Sequence σ_0 can be chosen such that $t_p \notin \sigma_0$. To prove this claim, assume that $\sigma_0 = \sigma_1 t_p \sigma_2$ with $t_p \notin \sigma_2$. Let $s'' \in \mathcal{B}(P)$ be the marking such that $(N, t_p{}^\bullet)\ [\sigma_1 t_p\rangle$ $(N, s'')\,[\sigma_2\rangle\,(N, s')$. It follows from Definition 2.3.7 (Firing rule) on Page 18 that $s'' \geq t_p{}^\bullet$. Since $(N, [p])$ is bounded, it follows from Property 2.3.63 on Page 32 that also $(N, t_p{}^\bullet)$ is bounded. Thus, Property 2.3.23 (Characterization of boundedness) on Page 23 yields that $s'' = t_p{}^\bullet$. Consequently, σ_2 is a firing sequence, satisfying the desired property that $t_p \notin \sigma_2$ and $(N, t_p{}^\bullet)\ [\sigma_2\rangle\ (N, s')$. Therefore, assume that $t_p \notin \sigma_0$. Since $s \geq [p]$, it follows that $(N_1, s-[p]\uplus t_p{}^\bullet)[\sigma_0\rangle(N_1, s-[p]\uplus s')$. Since $(N, s')[t\rangle$, it follows that $(N_1, s-[p]\uplus s')[t\rangle$. From $(N_1, [i])\ [\sigma t_p\rangle\ (N_1, s - [p] \uplus t_p{}^\bullet)$, it follows that $(N_1, [i])\ [\sigma t_p \sigma_0\rangle\ (N_1, s - [p] \uplus s')$. As a result, t is not dead in $(N_1, [i])$. Combining the results of the two cases yields that $(N_1, [i])$ has no dead transitions, which proves Requirement *vii)* of Definition 4.3.2 and completes the proof of Theorem 4.3.40. □

Example 4.3.42. Consider again Example 4.3.15 on Page 166. The relationship between $(N_2, [i])$ and $(N_1, [i])$ can be proven by means of transformation rule *PJ* given in Theorem 4.3.40 on Page 182. The transformation is illustrated in Figure 4.3.41 on the following page. The arc in P/T net N_1 between transition t_p and its output place p is replaced by net N minus the dashed arc between p and t_p. The result is P/T net N_2. Since $(N, [p])$ is free-choice, live, and bounded and since method identifier *cerr* does not occur as a transition label in N_1, $(N_2, [i])$ is a subclass under projection inheritance of $(N_1, [i])$.

To formulate the last transformation rule of this subsection, the following auxiliary definition is needed. A place of a marked P/T net is said to be *redundant* or *implicit* if and only if it does not depend on the number of tokens in the place whether any of its output transitions is enabled by some reachable marking.

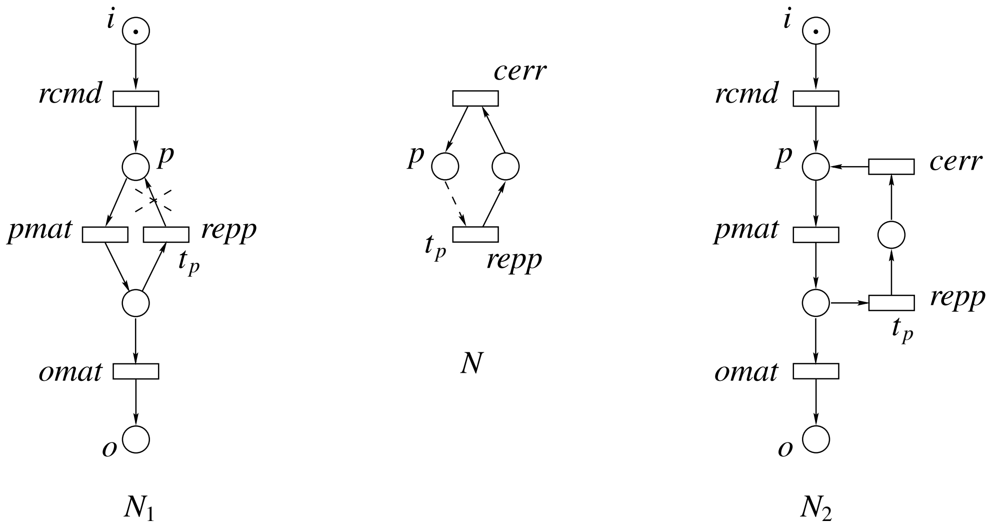


Figure 4.3.41: A transformation based on transformation rule PJ.

Definition 4.3.43. (Implicit place) Let (N, s) with $N = (P, T, F, \ell)$ be a marked, labeled, ordinary P/T net. A place $p \in P$ is called *implicit* in (N, s) if and only if, for all reachable markings $s' \in [N, s\rangle$ and transition $t \in p^\bullet$, $s' \geq {}^\bullet t \backslash \{p\} \Rightarrow s' \geq {}^\bullet t$.

Implicit places and their properties have been studied in (BERTHELOT, 1987; COLOM AND SILVA, 1990). In the context of this chapter, only the following property of implicit places is important. Implicit places different from the output place may be removed from an object life cycle yielding a P/T net which is branching bisimilar to the original life cycle. It is even possible to prove a stronger result, namely that the resulting P/T net is again an object life cycle. However, since this stronger result is not needed, only the first mentioned result is proven.

Lemma 4.3.44. *Let* $(N, [i])$ *with* $N = (P, T, F, \ell)$ *be an object life cycle containing a place* $p \in P \backslash \{o\}$ *that is implicit in* $(N, [i])$*. If* N_0 *is the subnet of* N *generated by the set of nodes* $P \backslash \{p\} \cup T$*, as defined in Definition 2.3.48 on Page 28, then* $(N, [i]) \sim_b (N_0, [i])$*.*

Proof. Let $\mathcal{R}$ be the relation $\{((N, u), (N_0, v)) \mid u \in [N, [i]\rangle \wedge v \in [N_0, [i]\rangle \wedge u \restriction (P \backslash \{p\}) = v\}$. It is not difficult to verify that, since p is implicit in $(N, [i])$, $p \neq i$. As a result, $(N, [i])\mathcal{R}(N_0, [i])$. It remains to be shown that $\mathcal{R}$ is a branching bisimulation. Assume that $u \in [N, [i]\rangle$ and $v \in [N_0, [i]\rangle$ are markings such that $(N, u)\mathcal{R}(N_0, v)$.

i) Assume that $t \in T$ is such that $(N, u)[t\rangle$ and $(N, u)\,[\ell(t)\rangle\,(N, u')$ with $u' = u - {}^\bullet t \uplus t^\bullet$. It follows immediately from the fact that $u \restriction (P \backslash \{p\}) = v$, the definition of N_0, and Definition 2.3.7 (Firing rule) on Page 18 that $(N_0, v)\ [\ell(t)\rangle\ (N_0, u' \restriction (P \backslash \{p\}))$. Hence, it easily follows that Requirement *i*) of Definition 2.2.15 (Branching bisimilarity) on Page 11 is satisfied.

ii) Assume that $t \in T$ is such that $(N_0, v)[t\rangle$ and $(N_0, v)\ [\ell(t)\rangle\ (N_0, v')$ with $v' = v - {}^\bullet t \uplus t^\bullet$. It follows from the facts that $u \restriction (P \backslash \{p\}) = v$ and that p is implicit in

$(N, [i])$, Definition 4.3.43 (Implicit place) on the preceding page, the definition of N_0, and Definition 2.3.7 (Firing rule) that $(N, u) [\ell(t)\rangle (N, v' \uplus ((u - {}^{\bullet}t \uplus t^{\bullet}) \lceil \{p\}))$. Thus, it is not difficult to verify that also Requirement *ii*) of Definition 2.2.15 is satisfied.

iii) First, assume that $\downarrow(N, u)$. It follows from Definition 4.3.1 (Operational semantics) on Page 160 that $u = [o]$. Since $u \lceil (P\backslash\{p\}) = v$ and $p \neq o$, also $v = [o]$. Thus, $\downarrow(N_0, v)$, which, by Definition 2.2.14 ($\Downarrow$) on Page 11, implies that $\Downarrow(N_0, v)$. Second, assume that $\downarrow(N_0, v)$. It follows that $v = [o]$. Since $u \lceil (P\backslash\{p\}) = v$ and $p \neq o$, $u = [o] \uplus (u \lceil \{p\})$. Since $(N, [i])$ is an object life cycle, it follows from Requirement *v*) (Proper termination) of Definition 4.3.2 (Object life cycle) on Page 161 and the fact that $u \in [N, [i]\rangle$ that $u = [o]$. As a result, $\Downarrow(N, u)$, which implies that also Requirement *iii*) of Definition 2.2.15 is satisfied. □

Lemma 4.3.44 on the facing page does not carry over to arbitrary ordinary P/T nets, as the reader familiar with the literature on implicit places might expect. The reason is the specific operational semantics for ordinary P/T nets chosen in this chapter. The termination predicate in this semantics says that a P/T net terminates if and only if it reaches a marking with a single token in output place o. To illustrate the problem, consider a P/T net which reaches a marking with a single token in o plus some additional tokens in an implicit place. Clearly, according to the semantics, this net has not terminated successfully. However, the P/T net that results from removing the implicit place does terminate in the corresponding marking, which means that the two nets cannot be branching bisimilar.

The final result of this subsection is the fourth transformation rule. It is inspired by and carries the same name as the algebraic axiom of inheritance *PJ*3, given in Property 4.2.27 on Page 157. Transformation rule *PJ*3 is formalized in Theorem 4.3.46 given below. It shows under what restrictions it is allowed to extend an object life cycle with a parallel branch of behavior. The result of rule *PJ*3 is a subclass of the original life cycle under projection inheritance. It is illustrated in Figure 4.3.45 on the next page. As before, $(N_0, [i])$ is the original life cycle. Again, the modification of N_0 is based on a free-choice P/T net N containing a place p such that $(N, [p])$ is live and bounded. The two net structures N_0 and N share two transitions t_i and t_o. In N, place p is the only input place of t_i and the only output place of t_o. Furthermore, p has no other input or output transitions. The net structure N_1 resulting from transformation rule *PJ*3 is defined as the union of N_0 and N after the removal of place p. These assumptions mean that transitions t_i and t_o function as the input and output transition of the extra parallel branch modeled by N. The basic idea is that the P/T net $(N_1, [i])$ satisfies the property that every firing of transition t_i is eventually followed by a firing of transition t_o. The requirement that $(N, [p])$ is free-choice, live, and bounded guarantees that each time transition t_i fires the resulting tokens in places of N can be moved to the input places of transition t_o in N by only firing transitions of N other than t_i and t_o. In addition, to guarantee that $(N_1, [i])$ satisfies the desired property, also $(N_0, [i])$ must be such that every firing of t_i is followed by exactly one firing of t_o. To achieve this goal, assume that N_0 is extended with a place q with t_i as its only input place and t_o as is only output place. Requiring that place q is implicit in this extension guarantees that a firing of transition t_o is always preceded by a firing of t_i. It is not difficult to see that the number of tokens in q corresponds to the number of firings of transition t_i which have not yet been followed by a firing of t_o. To guarantee that $(N_0, [i])$ cannot terminate without firing t_o as many times as t_i, the extension of N_0 with place q must be such that it cannot put a token in place o while leaving tokens in q. Clearly, this is achieved when the extension of N_0 with q yields another life cycle. The combination of the requirements on N_0 and N implies that $(N_1, [i])$ is an object life cycle satisfying the property that every firing of t_i is eventually followed by a firing of t_o.

The generality of transformation rule $PJ3$ is similar to the generality of rules PP and PJ. However, the requirements of $PJ3$ cannot be verified with the same efficiency. The two requirements concerning the extension of N_0 with place q are hard to verify. However, the requirement that q is implicit in the extension of N_0 can be slightly strengthened by requiring that q is *structurally* implicit (BERTHELOT, 1987; COLOM AND SILVA, 1990). This hardly compromises the generality of the rule. However, it is possible to verify whether a place is structurally implicit in polynomial time (COLOM AND SILVA, 1990). The requirement that the extension of N_0 with q yields another life cycle is, in general, also hard to verify. However, the input that $(N_0, [i])$ is a life cycle and that q is implicit in the extension might alleviate this problem. The applicability of the rule has to be tested in practice. It might also be an interesting question for future research to find the exact complexity of this problem.

Finally, because the proof of Theorem 4.3.46 is similar to the proof of Theorem 4.3.40 on Page 182, it is not given in as much detail and with the same amount of explanation as the proof of Theorem 4.3.40. However, all essential aspects of the proof are given in detail.

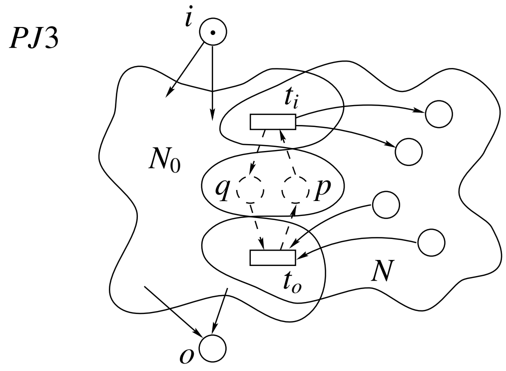


Figure 4.3.45: A projection-inheritance-preserving transformation rule.

Theorem 4.3.46. (Projection-inheritance-preserving transformation rule $PJ3$**)** *Let* $(N_0, [i])$ *with* $N_0 = (P_0, T_0, F_0, \ell_0)$ *be an object life cycle in* $\mathcal{L}$. *Let* $N = (P, T, F, \ell)$ *be a connected, labeled, free-choice P/T net. Assume that* $q \in U$ *is a fresh identifier not appearing in* $P_0 \cup T_0 \cup P \cup T$. *If* N *contains a place* $p \in P$ *and transitions* $t_i, t_o \in T$ *such that*

i) $^\bullet p = \{t_o\}$, $p^\bullet = \{t_i\}$, $^\bullet t_i = \{p\}$, $t_0{}^\bullet = \{p\}$,

ii) $p \notin \{i, o\}$, $P_0 \cap P = \emptyset$, $T_0 \cap T = \{t_i, t_o\}$,
iii) $(\forall t : t \in T \backslash T_0 : \ell(t) \notin \alpha(N_0, [i]))$,
iv) $(N, [p])$ *is live and bounded,*
v) $N_1 = N_0 \cup (P \backslash \{p\}, T, F \backslash \{(p, t_i), (t_o, p)\}, \ell)$ *is well defined,*
vi) q *is implicit in* $(N_0^q, [i])$ *with* $N_0^q = (P_0 \cup \{q\}, T_0, F_0 \cup \{(t_i, q), (q, t_o)\}, \ell_0)$, *and*
vii) $(N_0^q, [i])$ *is an object life cycle,*

then $(N_1, [i])$ *is an object life cycle in* $\mathcal{L}$ *such that* $(N_1, [i]) \leq_{pj} (N_0, [i])$.

Proof. The proof goes along the same lines as the proof of Theorem 4.3.40 on Page 182. Let $I = \alpha(N_1, [i]) \backslash \alpha(N_0, [i])$. It must be shown that $\tau_I(N_1, [i])$ is branching bisimilar to $(N_0, [i])$. According to Lemma 4.3.44 on Page 188, it suffices to show that $\tau_I(N_1, [i])$ is branching bisimilar to $(N_0^q, [i])$. Let $^\circ\_$, $\_^\circ$ denote the preset and postset of N_0^q, respectively. Let $\mathcal{R} = \{(\tau_I(N_1, u), (N_0^q, v)) \mid u \in [N_1, [i]\rangle \wedge v \in [N_0^q, [i]\rangle \wedge u \upharpoonright P_0 = v \upharpoonright P_0 \wedge [p^{v(q)}]$ is a home marking of $(N, u \upharpoonright P)\}$. Lemma 4.3.33 on Page 177 yields that $\tau_I(N_1, [i]) \mathcal{R} (N_0^q, [i])$. To prove that $\mathcal{R}$ is a branching bisimulation, assume that $u \in [N_1, [i]\rangle$ and $v \in [N_0^q, [i]\rangle$ are such that $\tau_I(N_1, u) \mathcal{R} (N_0^q, v)$. That is, $u \upharpoonright P_0 = v \upharpoonright P_0$ and $[p^{v(q)}]$ is a home marking of $(N, u \upharpoonright P)$.

i) Assume that $t \in T_0 \cup T$ is such that $\tau_I(N_1, u)[t\rangle$ and $\tau_I(N_1, u)\ [\alpha\rangle\ \tau_I(N_1, u')$ with $\alpha = (\ell_0 \cup \ell)(t)$ and $u' = u - {}^\bullet t \uplus t^\bullet$. It must be shown that there exist $v', v'' \in \mathcal{B}(P_0 \cup \{q\})$ such that *a)* $(N_0^q, v) \Longrightarrow (N_0^q, v'')\ [(\alpha)\rangle\ (N_0^q, v')$, *b)* $\tau_I(N_1, u) \mathcal{R} (N_0^q, v'')$, and *c)* $\tau_I(N_1, u') \mathcal{R} (N_0^q, v')$.
First, assume that $t \in T \backslash T_0$. The argument in this case is identical to the argument used in the proof of the analogue case of Theorem 4.3.40.
Second, assume that $t \in T_0$. Let $v'' = v$ and $v' = v - {}^\circ t \uplus t^\circ$.
To prove Requirement *a)*, observe that ${}^\circ t = {}^\bullet t \cap P_0$, unless $t = t_o$. In the latter case, ${}^\circ t = ({}^\bullet t \cap P_0) \cup \{q\}$. Requirement *a)* easily follows from the assumptions and the fact that q is implicit in $(N_0^q, [i])$.
Requirement *b)* follows immediately from the assumptions.
Finally, Requirement *c)* is proven as follows. It follows from the assumptions and Requirement *a)* that $u' \in [N_1, [i]\rangle$ and $v' \in [N_0^q, [i]\rangle$. The fact that $u \upharpoonright P_0 = v \upharpoonright P_0$ yields that $u' \upharpoonright P_0 = v' \upharpoonright P_0$. It remains to be shown that $[p^{v'(q)}]$ is a home marking of $(N, u' \upharpoonright P)$. Recall that $[p^{v(q)}]$ is a home marking of $(N, u \upharpoonright P)$.
Three cases can be distinguished. First, assume that $t \notin \{t_i, t_o\}$. It follows easily from the assumptions that $v'(q) = v(q)$ and $u' \upharpoonright P = u \upharpoonright P$, which immediately proves the requirement. Second, assume that $t = t_i$. It follows from the requirements of Theorem 4.3.46 that $v'(q) = v(q) + 1$ and $u' \upharpoonright P = u \upharpoonright P \uplus t^\cdot$. It follows from the fact that $[p^{v(q)}]$ is a home marking of $(N, u \upharpoonright P)$ and Property 2.3.66 (Monotonicity of home markings) on Page 32 that $[p^{v(q)+1}]$ is a home marking of $(N, u \upharpoonright P \uplus [p])$. Since ${}^\cdot t_i = \{p\}$, it follows from the assumption that $t = t_i$ and Definitions 2.3.7 (Firing rule) on Page 18 and 2.3.31 (Home marking) on Page 25 that $[p^{v(q)+1}]$ is a home marking of $(N, u \upharpoonright P \uplus t^\cdot)$, which proves the requirement in this case. Third,

assume that $t = t_o$. It follows that $v'(q) = v(q)-1$ and $u'\lceil P = u\lceil P - {}^{\cdot}t$. Definition 2.3.7 (Firing rule) yields that $(N, u \lceil P)\ [\ell(t)\rangle\ (N, u \lceil P - {}^{\cdot}t \uplus [p])$. The fact that $[p^{v(q)}]$ is a home marking of $(N, u\lceil P)$ and Definition 2.3.31 (Home marking) imply that $[p^{v(q)}]$ is a home marking of $(N, u\lceil P - {}^{\cdot}t \uplus [p])$. It follows from Lemma 4.3.37 on Page 179 that $[p^{v(q)-1}]$ is a home marking of $(N, u \lceil P - {}^{\cdot}t)$, which proves the requirement also in this case.

ii) Assume that $t \in T_0$ is such that $(N_0, v)[t\rangle$ and $(N_0, v)\ [\alpha\rangle\ (N_0, v')$ with $\alpha = \ell_0(t)$ and $v' = v'' - {}^{\circ}t \uplus t^{\circ}$. It must be shown that there exist $u', u'' \in \mathcal{B}(P_0 \cup P\backslash\{p\})$ such that *a*) $\tau_I(N_1, u) \Longrightarrow \tau_I(N_1, u'')\,[(\alpha)\rangle\,\tau_I(N_1, u')$, *b*) $\tau_I(N_1, u'')\mathcal{R}(N_0^q, v)$, and *c*) $\tau_I(N_1, u')\mathcal{R}(N_0, v')$. Two cases need to be distinguished.

First, assume $t \neq t_o$. Let $u'' = u$ and $u' = u - {}^{\bullet}t \uplus t^{\bullet}$. It follows from the assumptions that ${}^{\bullet}t = {}^{\circ}t$, which proves Requirement *a*). Requirement *b*) follows immediately from the assumptions. To prove Requirement *c*), observe that $v' \in [N_0^q, [i]\rangle$ and $u' \in [N_1, [i]\rangle$. Two cases are distinguished. If $t \neq t_i$, then $t^{\bullet} = t^{\circ}$. Thus, $u' \lceil P_0 = v'\lceil P_0$, $v'(q) = v(q)$, and $u'\lceil P = u\lceil P$. Requirement *c*) follows immediately. If $t = t_i$, then $t^{\bullet} = t^{\circ}\backslash\{q\} \cup t^{\cdot}$. It follows that $u'\lceil P_0 = v'\lceil P_0$, $v'(q) = v(q)+1$, $u'\lceil P = u\lceil P \cup t^{\cdot}$. It remains to be shown that $[p^{v'(q)}]$ is a home marking of $(N, u'\lceil P)$. It follows from the assumption that $[p^{v(q)}]$ is a home marking of $(N, u\lceil P)$ and Property 2.3.66 (Monotonicity of home markings) on Page 32 that $[p^{v(q)+1}]$ is a home marking of $(N, u\lceil P \uplus [p])$. Since $(N, u\lceil P \uplus [p])\,[\ell(t_i)\rangle\,(N, u\lceil P \uplus t^{\cdot})$, Definition 2.3.31 (Home marking) on Page 25 yields that $[p^{v(q)+1}]$ is a home marking of $(N, u\lceil P \uplus t^{\cdot})$, which proves the requirement.

Second, assume $t = t_o$. Let $u'' = v \lceil P_0 \uplus (\uplus r : r \in {}^{\cdot}t : [r^{v(q)}])$ and $u' = v' \lceil P_0 \uplus (\uplus r : r \in {}^{\cdot}t : [r^{v'(q)}])$. Since $t = t_o$, it follows that ${}^{\bullet}t = {}^{\circ}t\backslash\{q\} \cup {}^{\cdot}t$ and $t^{\bullet} = t^{\circ}$. The proof of Requirement *a*) is as follows. The fact that $[p^{v(q)}]$ is a home marking of $(N, u\lceil P)$ and Lemma 4.3.38 on Page 180 imply that there exists a firing sequence $\sigma \in (T\backslash\{t_i\})^*$ such that $(N, u \lceil P)\ [\sigma\rangle\ (N, [p^{v(q)}])$. Note that it follows from the definition of N_1 that $u(p) = 0$. Consequently, σ can be chosen such that $\sigma = \sigma_0\sigma_1$ where $t_o \notin \sigma_0$ and where σ_1 is of length $v(q)$ with, for all i, $0 \leq i < v(q)$, $\sigma_1(i) = t_o$. As a result, $(N, u\lceil P)\,[\sigma_0\rangle\,(N, (\uplus r : r \in {}^{\cdot}t : [r^{v(q)}])$. It follows from the fact that $u \lceil P_0 = v \lceil P_0$ that $\tau_I(N_1, u)\ [\sigma_0\rangle\ \tau_I(N_1, u'')$. Requirement *iii*) of Theorem 4.3.46 yields that $\tau_I(N_1, u) \Longrightarrow \tau_I(N_1, u'')$. It follows from the observation that ${}^{\bullet}t = {}^{\circ}t\backslash\{q\} \cup {}^{\cdot}t$, $t^{\bullet} = t^{\circ}$, and $v'(q) = v(q) - 1$ that $\tau_I(N_1, u'')\ [\ell_0(t)\rangle\ \tau_I(N_1, u')$, which proves Requirement *a*). To prove Requirement *b*), observe that $u'' \lceil P_0 = v \lceil P_0$. It remains to be shown that $[p^{v(q)}]$ is a home marking of $(N, u'' \lceil P)$. In the proof of Requirement *a*), it has been shown that $u'' \lceil P \in [N, u \lceil P\rangle$. Thus, the fact that $[p^{v(q)}]$ is a home marking of $(N, u \lceil P)$ implies that $[p^{v(q)}]$ is also a home marking of $(N, u'' \lceil P)$. To prove Requirement *c*), note that $u' \lceil P_0 = v' \lceil P_0$. Therefore, it suffices to show that $[p^{v'(q)}]$ is a home marking of $(N, u' \lceil P)$. Observe that $v'(q) = v(q)-1$ and $u'\lceil P = u'' - {}^{\cdot}t$. Since $(N, u''\lceil P)[\ell(t)\rangle(N, u''\lceil P - {}^{\cdot}t \uplus [p])$, it follows from the fact that $[p^{v(q)}]$ is a home marking of $(N, u'' \lceil P)$ that also $[p^{v(q)}]$ is a home marking of $(N, u''\lceil P - {}^{\cdot}t \uplus [p])$. Thus, Lemma 4.3.37 on Page 179 implies that $[p^{v(q)-1}]$ is a home marking of $(N, u'' \upharpoonright P - {}^{\bullet}t)$, which proves Requirement c).

iii) The argument to prove Requirement *iii* of Definition 2.2.15 (Branching bisimilarity) on Page 11 is identical to the argument used in the corresponding case of the proof of Theorem 4.3.40 on Page 182.

It remains to be shown that $(N_1, [i])$ is an object life cycle. It is clear that $(N_1, [i])$ satisfies Requirements *i*) through *iv*) of Definition 4.3.2 (Object life cycle) on Page 161. The proof that $(N_1, [i])$ satisfies Requirement *v*) (Proper termination) of Definition 4.3.2 is almost identical to the corresponding part of the proof of Theorem 4.3.40.

To prove that $(N_1, [i])$ satisfies Requirement *vi*) (Termination option) of Definition 4.3.2, assume that $u \in [N_1, [i]\rangle$. Since $\mathcal{R}$ is a branching bisimulation, Lemma 4.3.17 on Page 168 yields that there exists a $v \in [N_0^q, [i]\rangle$ such that $u \upharpoonright P_0 = v \upharpoonright P_0$ and $[p^{v(q)}]$ is a home marking of $(N, u \upharpoonright P)$. As shown in part *ii*) of the proof that $\mathcal{R}$ is a branching bisimulation, it is possible to construct a firing sequence $\sigma \in (T\backslash\{t_i, t_o\})^*$ such that $(N_1, u)\ [\sigma\rangle\ (N_1, v \upharpoonright P_0 \uplus (\uplus r : r \in {}^{\bullet}t_o : [r^{v(q)}]))$. It follows from the fact that $(N_0^q, [i])$ is a life cycle that there is a firing sequence $\sigma_0 \in T_0^*$ such that $(N_0^q, v)\ [\sigma_0\rangle\ (N_0^q, [o])$. It is possible to transform σ_0 into a sequence $\bar{\sigma_0} \in (T_0 \cup T)^*$ such that $(N_1, v \upharpoonright P_0 \uplus (\uplus r : r \in {}^{\bullet}t_o : [r^{v(q)}]))\ [\bar{\sigma_0}\rangle\ (N_1, [o])$, proving that $(N_1, [i])$ satisfies Requirement *vi*) of Definition 4.3.2. The transformation is inductively defined as follows. If $t_i \notin \sigma_0$, then $\bar{\sigma_0} = \sigma_0$. It is straightforward to see that this firing sequence satisfies the requirement. If $t_i \in \sigma_0$, then assume that $\sigma_0 = \sigma_1 t_i \sigma_2$ with $t_i \notin \sigma_1$. Let $v', v'' \in \mathcal{B}(P_0 \cup \{q\})$ be such that $(N_0^q, v)\ [\sigma_1\rangle\ (N_0^q, v')\ [t_i\rangle\ (N_0^q, v'')\ [\sigma_2\rangle\ (N_0^q, [o])$. It follows that $(N_1, v \upharpoonright P_0 \uplus (\uplus r : r \in {}^{\bullet}t_o : [r^{v(q)}]))\ [\sigma_1\rangle\ (N_1, v' \upharpoonright P_0 \uplus (\uplus r : r \in {}^{\bullet}t_o : [r^{v'(q)}]))\ [t_i\rangle\ (N_1, v'' \upharpoonright P_0 \uplus (\uplus r : r \in {}^{\bullet}t_o : [r^{v'(q)}]) \uplus t_i{}^{\bullet})$. Lemma 4.3.34 on Page 177 states that $[p]$ is a home marking of $(N, [p])$. Since $(N, [p])[\ell(t_i)\rangle(N, t_i{}^{\bullet})$, it follows that $[p]$ is also a home marking of $(N, t_i{}^{\bullet})$. Lemma 4.3.38 on Page 180 and the fact that ${}^{\bullet}p = \{t_o\}$ imply that there exists a firing sequence $\sigma_3 \in (T\backslash\{t_i, t_o\})^*$ such that $(N, t_i{}^{\bullet})\,[\sigma_3\rangle\,(N, {}^{\bullet}t_o)$. It follows that $(N_1, v''\upharpoonright P_0 \uplus (\uplus r : r \in {}^{\bullet}t_o : [r^{v'(q)}]) \uplus t_i{}^{\bullet})\,[\sigma_3\rangle\,(N_1, v''\upharpoonright P_0 \uplus (\uplus r : r \in {}^{\bullet}t_o : [r^{v'(q)+1}]))$. Observe that $v''(q) = v'(q)+1$. Hence, $(N_1, v \upharpoonright P_0 \uplus (\uplus r : r \in {}^{\bullet}t_o : [r^{v(q)}]))\,[\sigma_1 t_i \sigma_3\rangle\,(N_1, v'' \upharpoonright P_0 \uplus (\uplus r : r \in {}^{\bullet}t_o : [r^{v''(q)}]))$. Choosing $\bar{\sigma_0} = \sigma_1 t_i \sigma_3 \bar{\sigma_2}$ yields that $(N_1, v \upharpoonright P_0 \uplus (\uplus r : r \in {}^{\bullet}t_o : [r^{v(q)}]))\ [\bar{\sigma_0}\rangle\ (N_1, [o])$, completing the proof.

The proof that $(N_1, [i])$ also satisfies Requirement *vii*) of Definition 4.3.2 is very similar to the corresponding part of the proof of Theorem 4.3.40. The only differences are that t_i plays the role of t_p and that the transformation of firing sequences is the one introduced in the previous part of the proof. □

Example 4.3.48. Let us return one more time to Example 4.3.15 on Page 166. The subclass relationship between $(N_3, [i])$ and $(N_2, [i])$ can be proven by means of transformation rule *PJ*3 of Theorem 4.3.46 on Page 190. The transformation is depicted in Figure 4.3.47 on the next page. Let N_2^q denote the P/T net N_2 extended with place q. According to Definition 4.3.43 on Page 188, place q is implicit in $(N_2^q, [i])$. In addition, $(N_2^q, [i])$ is an object life cycle. This means that the extension of N_2 with place q satisfies the requirements of Theorem 4.3.46. As explained, these requirements guarantee that, in

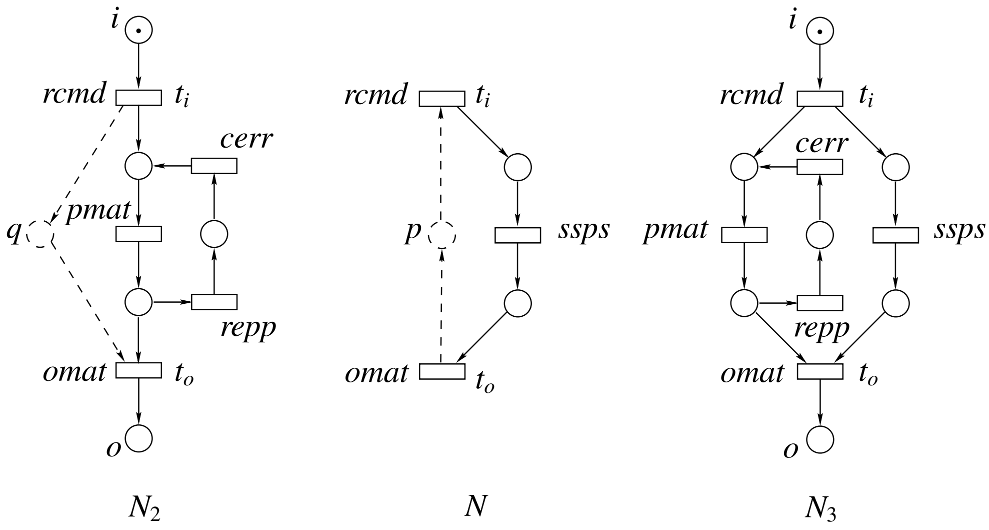


Figure 4.3.47: A transformation based on transformation rule *PJ*3.

$(N_2, [i])$, each firing of transition t_i is ultimately followed by exactly one firing of transition t_o. The transformation of N_2 is based on the connected, free-choice P/T net N. Since $(N, [p])$ is live and bounded and since *ssps* is a method identifier not occurring in N_2, also N satisfies the requirements of Theorem 4.3.46. The union of N_2 and N minus place p and its incoming and outgoing arcs yields P/T net N_3. Thus, it may be concluded from Theorem 4.3.46 that object life cycle $(N_3, [i])$ is a subclass under projection inheritance of life cycle $(N_2, [i])$.

So far in this subsection, four transformation rules have been presented to construct subclasses of object life cycles under different forms of inheritance. The rules correspond to design constructs that are often used in practice, namely choice, iteration, sequential composition, and parallel composition. As explained in the introduction to this subsection, the rules are compromises between generality and efficiency. Transformation rule *PT*, corresponding to the choice construct, is very general but not very efficient. Rules *PP* and *PJ*, corresponding to iteration and sequential composition respectively, are reasonably general and very efficient. Rule *PJ*3, corresponding to parallel composition, is reasonably general, but, at least theoretically, not very efficient. The applicability of all the rules has to be tested in practice. Nevertheless, it is interesting to briefly consider generalizations and restrictions of the rules.

Rules *PP*, *PJ*, and *PJ*3 are all transformations of the original life cycle based on a live and bounded free-choice P/T net. This raises the question whether there also exists such a rule corresponding to the choice construct. It appears that this is not the case without compromising the generality of the current rule *PT* too much.

Another interesting question is whether it is possible to drop the free-choice requirement on the extensions in rules *PP*, *PJ*, and *PJ*3. The free-choice requirement itself is not essential. It is important that the class of nets on which the extension in the transformation rules is based satisfies the various results about home markings derived in this subsection and Section 2.3.5 for live and bounded free-choice nets. The class of live and bounded *ordinary* P/T nets is too general for this purpose, as can be shown by means of the P/T net in Figure 4.3.49.

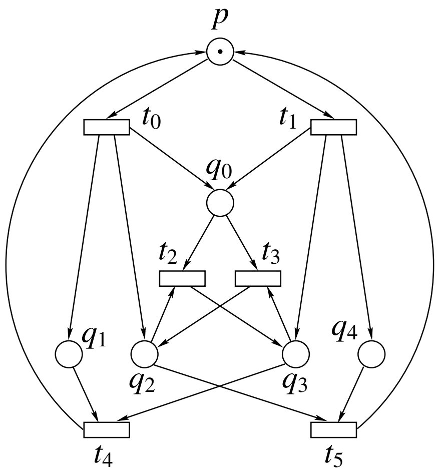


Figure 4.3.49: Non-monotonicity of home markings for live and bounded ordinary P/T nets.

The figure shows an ordinary P/T net N, which is not free-choice, with a place p containing a single token. It is not difficult to verify that $(N, [p])$ is live and bounded and that $[p]$ is a home marking of $(N, [p])$. However, consider the marked, ordinary P/T net $(N, [p^2])$. Firing sequence $t_0t_0t_2t_3$ leads to marking $[{q_1}^2, {q_2}^2]$. In this marking, no transitions are enabled. Clearly, this implies that $[p^2]$ is not a home marking of $(N, [p^2])$. Thus, the class of live and bounded ordinary P/T nets does not satisfy the property that home markings are monotone. Note that $(N, [p^2])$ is also no longer live and bounded. Since the reachable marking $[{q_1}^2, {q_2}^2]$ is a deadlock, $(N, [p^2])$ is obviously not live. Furthermore, firing the sequence of transitions $t_0t_1t_4t_5$, starting from the initial marking, yields marking $[p^2, {q_0}^2]$, which means that the number of tokens in place q_0 can increase indefinitely. Thus, $(N, [p^2])$ is also not bounded.

A consequence of the observation that the class of live and bounded ordinary P/T nets does not satisfy monotonicity of home markings is that it is not possible to omit the free-choice requirement in rules *PP*, *PJ*, and *PJ*3. It remains an interesting open problem to find classes of P/T nets that could replace the class of live and bounded free-choice P/T nets in the transformation rules. The class of live and bounded well-handled P/T nets might be a candidate. Well-handled P/T nets and some of their properties are studied in (VAN DER AALST, 1996, 1998).

Another option to generalize transformation rules *PP*, *PJ*, and *PJ*3 is to base the extension in the rules on the class of live and bounded ordinary P/T nets, dropping the free-choice requirement, while at the same time adding other restrictions to the rules to guarantee their correctness. Note that any such restrictions must exclude the P/T net of Figure 4.3.49 as an allowed extension. Requiring that the result of the transformation yields again a life cycle might be sufficient. However, proving such a generalization requires a lot of work. It is not clear whether it is worth the effort. The verification whether a particular transformation satisfies the requirements of such a generalized rule is very inefficient. As a result, it might be very hard to apply such a rule in practice. A second reason for not putting too much effort in generalizing the current set of transformation rules is that these rules have not yet been thoroughly tested in practice.

Another interesting observation about rules PP, PJ, and $PJ3$ is the following. Even if the original life cycle is free-choice, the resulting life cycle is not necessarily free-choice. As explained in Example 4.3.48 on Page 193, the inheritance relationship between life cycles $(N_2, [i])$ and $(N_3, [i])$ in Figure 4.3.14 on Page 167 can be proven by means of transformation rule $PJ3$. Clearly, N_2 is free-choice, whereas N_3 is not free-choice.

Comparing the transformation rules in this subsection to the axioms of inheritance in Section 4.2.2, it is noteworthy that no transformation rules have been given corresponding to the axioms of life-cycle inheritance defined in Property 4.2.30 on Page 158. The reason is that such transformation rules are simply combinations of the four rules of this subsection. All kinds of combinations of transformation rules are allowed, as long as the original life cycle and its extensions satisfy certain restrictions with respect to their transition labels. Basically, it must be guaranteed that methods which function as guards in an extension based on transformation rule PT do not appear in extensions based on any of the other transformation rules. This restriction follows from the original definition of the four inheritance relations. The guards in transformation rule PT are encapsulated, whereas the new methods in any of the extensions in the other transformation rules are hidden. As explained, in the definition of life-cycle inheritance, it is not allowed to treat different calls of the same methods in a different way. Examples of combinations of transformation rules are the following. Clearly, based on transformation rule PJ, it is allowed to replace two arcs in a P/T net simultaneously as long as in both cases the requirements of rule PJ are met. In this case, the abovementioned restriction is trivially satisfied. It is also allowed to replace an arc while at the same time adding an iteration. Another example is the simultaneous extension of a life cycle with an alternative and a parallel branch of behavior, which is allowed when the guards of the alternative do not appear in the extra parallel branch.

Finally, as already mentioned in Section 4.3.1, some of the rules developed in this subsection can also be used to construct object life cycles in general, without the aim to construct subclasses of existing life cycles. In particular, rules PP and PJ are suited for this purpose. The requirements of these rules can be efficiently verified and the result is again a life cycle. However, also rule $PJ3$ might be useful in some cases. Since transition labeling does not play a role in the definition of object life cycles, the rules can even be generalized when the aim is solely to construct object life cycles. The requirement that the labels of transitions added to the original object life cycle are method identifiers that do not yet appear in its alphabet may be dropped. An interesting consequence of the observation that the abovementioned transformation rules preserve life-cycle properties is that, based on Theorem 4.3.7 (Characterization of object life cycles) on Page 163, they can be easily transformed into liveness-and-boundedness-preserving transformation rules on ordinary P/T nets. Recall that also (VAN DER AALST, 1997) and (VALETTE, 1979) describe a set of liveness-and-boundedness-preserving transformation rules that, similar to the rules of this subsection, correspond to design constructs such as sequential composition, parallel composition, and iteration. However, the rules presented in this subsection are far more general than the rules of (VAN DER AALST, 1997) and (VALETTE, 1979). The fact that the transformation rules of this subsection can be transformed into liveness-and-boundedness-preserving transformation rules implies that they might have many more applications than only the one that is presented in this chapter.

4.4 Case Study: A Groupware Editor

4.4.1 Introduction

To validate the theory developed so far, the framework of the previous section is applied to a small case study, namely the development of a groupware editor. The Petri-net-framework is chosen instead of the process-algebraic-framework of Section 4.2, because the former is closer to practical applicability than the latter. The aim of this case study is not to give full specifications of all the classes in the design of a groupware editor, but to focus on the object life cycles. This means that data types, class attributes, and method implementations are omitted. If necessary, an informal explanation is given. The results in this section will also be published in (BASTEN AND VAN DER AALST, 1998).

Informal system requirements. The informal requirements for the groupware editor are as follows. The editor is meant to edit some kind of diagrams. Multiple users, possibly situated at different workstations, may be editing a single diagram in a joint editing session. Users may choose to either view or edit a diagram. They may join or leave a session at will. It must be clear to all users who is currently editing some given diagram. The diagrams under consideration may be complex, possibly consisting of multiple components. Components may introduce hierarchy in a diagram. It is possible to open or close a component revealing or hiding its details. Not just any user may view or edit any component in a diagram. Users must have permission to do so. Permissions are not fixed; to get permission, a user can simply select a component and then try the desired command. If there are no conflicting permissions, the command succeeds; otherwise, it fails and the user is notified. Users may explicitly surrender permissions. Most permissions are automatically reset if the user leaves the editing session; a few permissions may persist between sessions.

In the next subsections, the concept of life-cycle inheritance is used to design a groupware editor satisfying the abovementioned informal requirements. The development is split into three steps. First, a groupware *viewer* is designed. The design is fairly simple and the resulting system allows multiple users to view existing diagrams. Second, the viewer is transformed into a multi-user editor by adding editing functions to the life cycles of classes in the viewer design. Third, the multi-user editor is specialized to a groupware editor by adding permissions. The case study shows how life-cycle inheritance can be used to structure an object-oriented design process. It also shows how object life cycles can be reused in a design.

4.4.2 The multi-user viewer

Four classes can be distinguished in the design of the viewer: `user`, `diagram`, `user_session`, and `edit_session`. The first two classes have straightforward interpretations. The third and fourth class are intended to structure viewing sessions. During such a session, an `edit_session` object keeps track of all users viewing a particular diagram. An object of class `user_session` maintains all data involved in a particular session of a particular user. Figure 4.4.1 shows the life cycles for the objects of the above four classes.

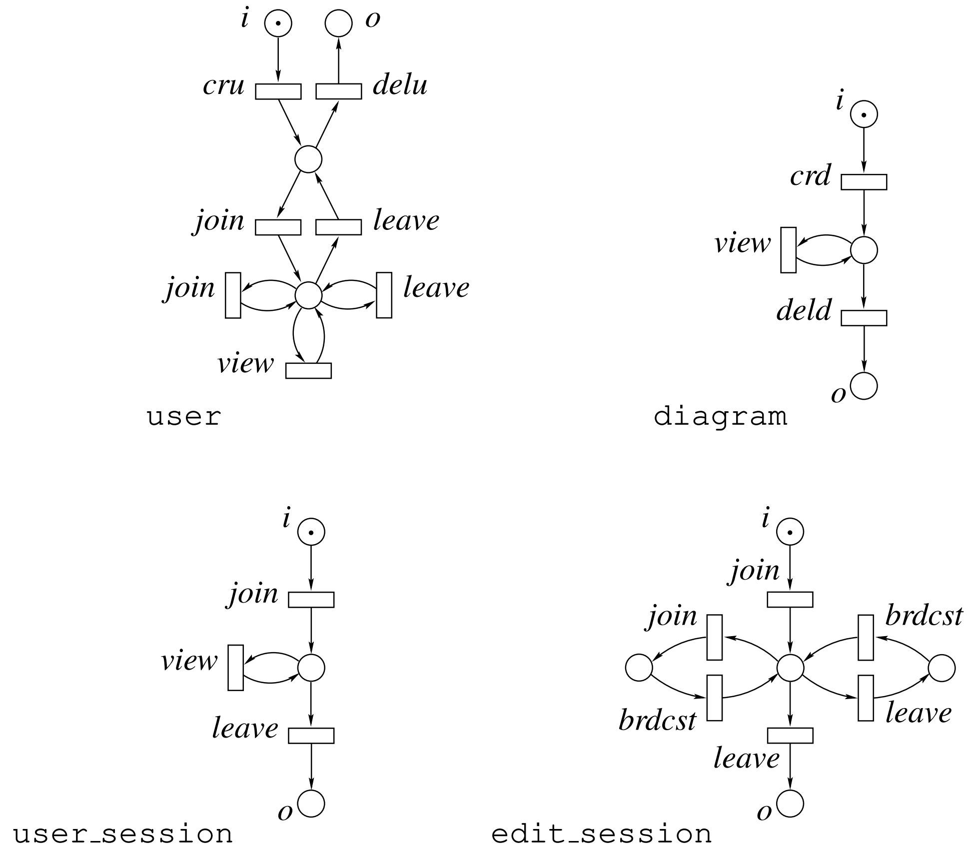


Figure 4.4.1: A multi-user viewer.

Users can be created (*cru*) or deleted (*delu*). This means that they are added to or removed from the list of users of the viewer. Once a user exists, he or she may join and leave edit sessions. A user that participates in at least one edit session may issue *view* commands in order to view diagrams. At this point, the user has no other commands available.

Class `diagram` has a very straightforward life cycle. Diagrams can be created (*crd*), viewed (*view*), and deleted (*deld*).

A user interacts with a diagram through a `user_session` object. Each time the user joins an edit session, a new object of class `user_session` is created. As before, the *view* command is the only possible command a user can execute. If the user leaves the edit session, the `user_session` is terminated.

Objects of class `edit_session` keep track of all users involved in a single edit session. The first user that "joins" an edit session for some diagram, actually creates a new `edit_session` object. If users join an already existing session, no new object is created; the information is simply stored in the existing object. To fulfill the requirement that all users must know who is participating in the session, it is assumed that the implementation of *join* is such that the new user gets a list of users already present. Furthermore, the information about a new user is broadcast (*brdcst*) to all other users participating in the session. When the user leaves, this information is broadcast to all remaining users. The last user leaving the session terminates the `edit_session` object.

In a complete object-oriented specification, it must be clear which methods interact with each other. In this case study, the interaction between methods is not formally specified. Instead, the following assumptions are made. First, `user` methods invoke methods with the same name in `user_session`, which, in turn, invoke methods of the same name in `edit_session` and `diagram`. Second, methods which do not have counterparts in one of the other classes are assumed to interact with (objects in) the environment. The examples of these methods in Figure 4.4.1 on the facing page are *cru*, *delu*, *crd*, *deld*, and *brdcst*.

4.4.3 The multi-user editor

In the second step of the design process, the multi-user viewer of the previous subsection is transformed into a multi-user editor. Permissions are not yet incorporated. They are added in the third and final step of the design. Consequently, in the version of the editor described in the current subsection, it is possible that a component is deleted by one user, while it is being changed or viewed by another one.

Three classes in the design of the viewer, shown in Figure 4.4.1 on the preceding page, are extended with editing facilities, namely `user`, `diagrams`, and `user_session`. Class `edit_session` does not need to change. Thus, the design of the editor consists of four classes, namely `user_e`, `diagram_e`, `user_session_e`, and `edit_session`. The new classes are shown in Figure 4.4.2 on the following page. In the remainder of this subsection, it is shown that the three new classes are subclasses of the corresponding classes of the viewer of Figure 4.4.1 under life-cycle inheritance. The places in life cycles `user_e` and `user_session_e` are marked with identifiers for the purpose of future reference.

First, consider class `diagram_e`. In the editor, it is possible to modify diagrams by means of method *mod*. It easily follows from transformation rule *PP* of Theorem 4.3.30 on Page 174 that `diagram_e` is a subclass of class `diagram` under protocol/projection inheritance. Thus, it also is a subclass of `diagram` under life-cycle inheritance.

Second, class `user` of Figure 4.4.1 has been extended to class `user_e` in the design

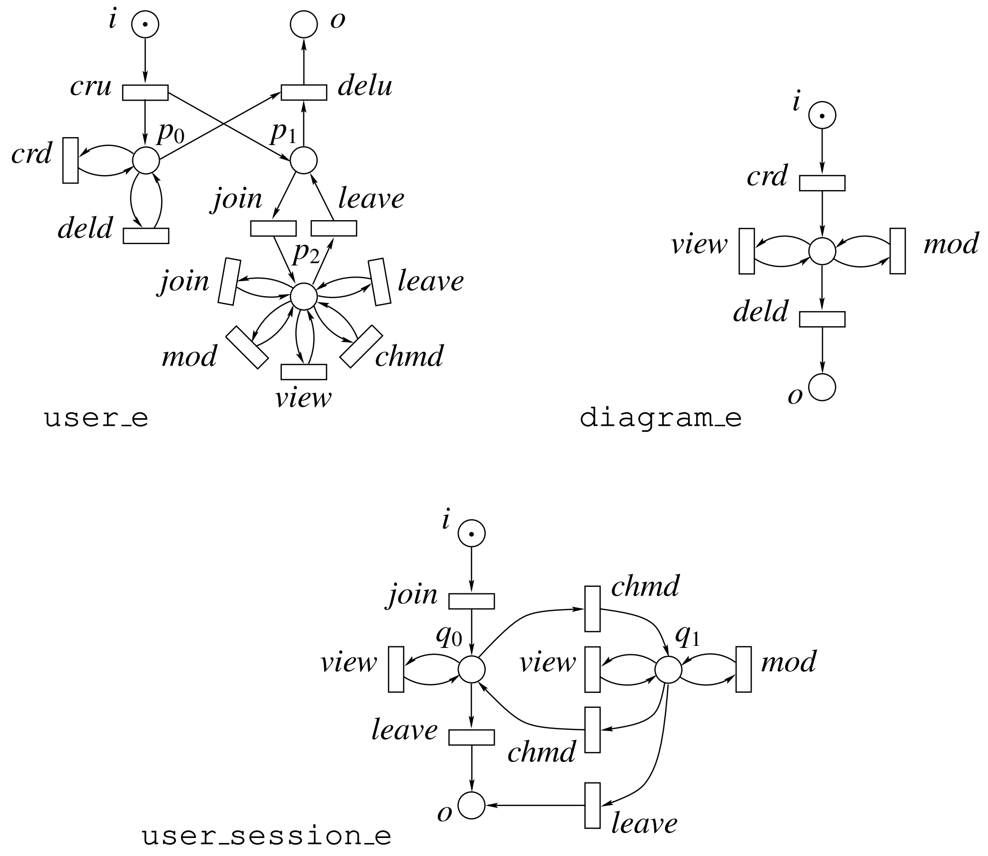


Figure 4.4.2: A multi-user editor.

of the editor. In the editor, users of the system are responsible for creating and deleting diagrams (methods *crd* and *deld*). Creation and deletion of diagrams can be done independently of editing (other) diagrams. Thus, a branch of parallel behavior, consisting of place p_0 and the two aforementioned methods, is added to the life cycle of class `user`. Transformation rule *PJ*3 of Theorem 4.3.46 on Page 190 can be used to prove that this extension yields a subclass under projection inheritance. The two transitions labeled *cru* and *delu* serve as the input and output transition of the additional parallel branch, respectively. Due to the presence of place p_1, any extra place between *cru* and *delu* is clearly implicit. For the same reason, the addition of such an extra place preserves the life cycle properties.

Another extra feature of the editor compared to the viewer of the previous subsection is that users are allowed to modify diagrams. For this purpose, class `user_e` contains method *mod*. Users have to choose whether they want to modify a diagram or whether they are satisfied with just the option to view it. Method *chmd* (change mode) can be used to toggle between viewing and editing mode. The implementation of methods *mod* and *chmd* can be such that only a subset of users is allowed to enter the editing mode. It simply follows from applying transformation rule *PP* that the addition of methods *mod* and *chmd* to class `user` leads to a subclass.

It is not difficult to verify that the subsequent application of transformation rules *PJ*3 and *PP* yields that `user_e` is a subclass of `user` under life-cycle inheritance.

Third, it must be shown that `user_session_e` is a subclass of `user_session`. For each edit session a user joins, a `user_session_e` object is created. This object keeps track of the mode in which the user is for this particular diagram. Initially, the user is in viewing mode. By invoking method *chmd*, the user can change to editing mode. This means that the object life cycle of class `user_session` in Figure 4.4.1 is extended with a choice. Transformation rule *PT* of Theorem 4.3.27 on Page 172 can be applied to show that `user_session_e` is a subclass of `user_session`. Method *chmd* acts as the guard.

Summarizing, three of the four transformation rules of the previous section have been used to extend the viewer of Figure 4.4.1 on Page 198 to an editor. The most important conclusion of this subsection is that the specifications of the object life cycles of the viewer have been *reused* during the design process.

4.4.4 The groupware editor

In the last step of the design of the groupware editor, a mechanism is added to handle permissions. Permissions are needed to prevent all kinds of anomalies. For example, they guarantee that it is impossible that one user deletes a component of a diagram when, at the same time, another user is viewing this component. Four new classes are designed, namely `user_p`, `diagram_p`, `user_session_p`, and `edit_session_p`. Figure 4.4.3 shows two of these four classes.

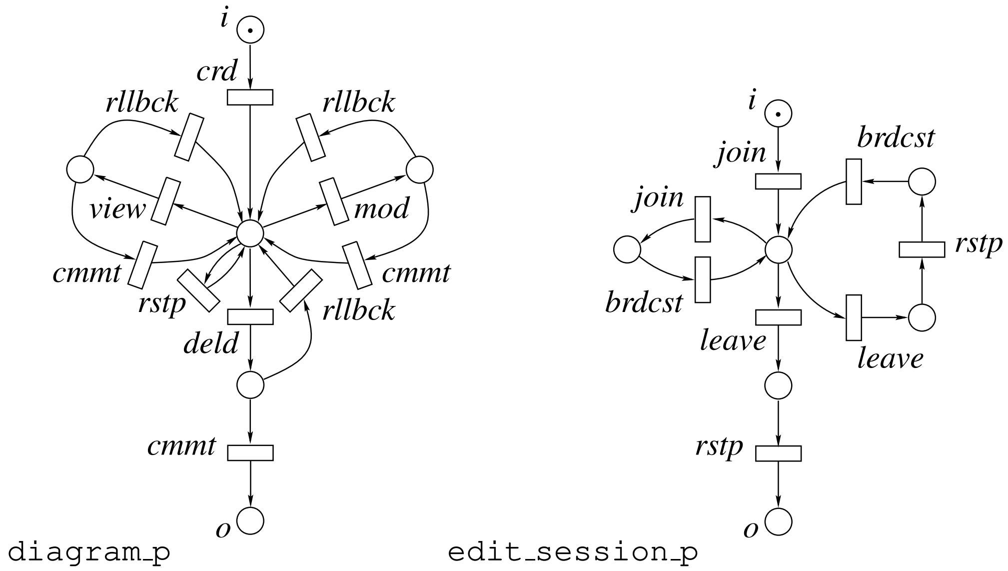


Figure 4.4.3: A groupware editor.

The current set of permissions for editing a diagram is maintained in the corresponding `diagram_p` object. This is the only feasible option since some permissions may persist between edit sessions, as stated in the system requirements explained in Section 4.4.1. Also explained in the requirements is that a user can get permission for some editing command by simply selecting a component and executing the command. Therefore, after every editing action *view*, *mod*, or *deld*, diagram_p executes either a rollback (*rllbck*) or a commit (*cmmt*), depending on whether or not the user has permission for the specific editing action. Another change in diagram_p when compared to class diagram_e is that the method *rstp* has been added which can be used to reset permissions. It must be taken care of that the three other classes invoke *rstp* whenever appropriate.

Classes user_p and user_session_p are simple extensions of classes user_e and user_session_e of Figure 4.4.2 on Page 200. Therefore, these two classes are not shown in Figure 4.4.3 on the preceding page. Since users may surrender permissions at any time, class user_p is obtained from user_e by adding a loop consisting of a single transition labeled *rstp* to place p_2. Since users interact with diagrams through user sessions, the life cycle of user_session_p is constructed from user_session_e by adding the same loop as above to places q_0 and q_1.

The fourth class in the design of the groupware editor is class edit_session_p which is constructed from class edit_session of Figure 4.4.1 on Page 198. As mentioned in Section 4.4.1, permissions are reset when a user leaves a session. This means that edit_session is extended with calls of method *rstp* after each invocation of the *leave* method.

It remains to be shown that the new classes are subclasses of the corresponding classes in the earlier designs. The addition of *rstp* to classes user_e and user_session_e is captured by transformation rule *PP* of Theorem 4.3.30 on Page 174. Hence, user_p and user_session_p are subclasses of user_e and user_session_e, respectively.

The addition of *rstp* to diagram_e is also captured by rule *PP*. To show that the addition of method *cmmt* to diagram_e is correct, a generalization of transformation rule *PJ* as discussed at the end of the previous section is needed. Applying rule *PJ* three times simultaneously shows that the extension with method *cmmt* yields a subclass. To show that the addition of method *rllbck* is allowed, consider the intermediate result obtained after the previous two additions. The desired result follows easily from transformation rule *PT* of Theorem 4.3.27 on Page 172. Hence, class diagram_p is a subclass of diagram_e.

Finally, to show that edit_session_p is a subclass of edit_session, it suffices to apply rule *PJ* twice simultaneously.

Summarizing the results of this subsection, it has been shown that the four classes in the design of the groupware editor are subclasses under life-cycle inheritance of the corresponding classes in the multi-user editor discussed in the previous subsection. The classes of the multi-user editor, in turn, are subclasses of the corresponding classes of the viewer presented in Section 4.4.2. Since life-cycle inheritance is transitive, it follows that the four classes of the groupware editor are also subclasses of the corresponding viewer classes. This means that the groupware editor preserves the behavior of the viewer. Although the case study is not very complex, it shows that the concept of life-cycle inheritance can be used to structure the design process. In addition, the transformation rules stimulate the reuse of object life cycles. Observe that all four transformation rules of Section 4.3.3 are needed in the case study. More realistic case studies are needed to test whether the four rules are sufficient in practical design situations. For this purpose, it is also necessary to incorporate the results of this chapter in a complete object-oriented formalism.

4.5 Conclusions

Concluding remarks. The concept of inheritance is one of the key concepts in object-oriented design. However, in most object-oriented methods which are in common use, such as OMT (RUMBAUGH ET AL., 1991), OOD (BOOCH, 1994), and UML (FOWLER AND SCOTT, 1997), inheritance is only well defined for the set of methods of a class and its attributes. It is implicitly assumed that the behavior of an object of a subclass extends the behavior of an object of its superclass. To overcome this omission in the definition of inheritance, this chapter studies inheritance of dynamic behavior.

Section 4.2 studies inheritance of dynamic behavior in a simple process-algebraic setting. Process algebra is particularly well suited for this purpose, because it does not have an explicit representation of process states. In addition, the notions of encapsulation and abstraction have been studied extensively in a process-algebraic setting. Encapsulation corresponds to *blocking* actions, whereas abstraction corresponds to *hiding* actions. Intuitively, blocking and hiding method calls play an important role in inheritance of dynamic behavior. Section 4.2.1 introduces four inheritance relations, based on either encapsulation or abstraction or both. Each of the inheritance relations captures different kinds of extensions to an object life cycle. The axioms of inheritance of Section 4.2.2 illustrate the most characteristic extensions allowed under each form of inheritance. Protocol inheritance captures the addition of an alternative to an object life cycle. Projection inheritance allows the addition of parallel and sequential behavior. Under protocol/projection inheritance, it is allowed to postpone behavior. In a setting which allows the specification of iterative behavior, protocol/projection inheritance captures the extension of an object life cycle with an iteration. Life-cycle inheritance, which is the most general form of inheritance discussed in this chapter, allows all kinds of combinations of the above mentioned extensions.

In Section 4.3, the four inheritance relations of Section 4.2 are studied in the framework of Petri nets. Due to the graphical nature of Petri nets and their explicit representation of states, the Petri-net framework is closer to object-oriented methods used in practice than the algebraic framework. Inspired by the algebraic axioms of inheritance, four transformation rules have been presented, which can be used to transform a class into a subclass. In Section 4.4, these transformation rules have been used in the development of a groupware editor. The concept of life-cycle inheritance and the accompanying transformation rules prove to be useful in structuring the design process. In addition, they stimulate the reuse of object life cycles. The most important conclusion is that the concepts developed in this chapter can be a promising addition to existing object-oriented methods.

The development of the concept of inheritance of dynamic behavior in this chapter is a perfect illustration of the complementary nature of process algebra and Petri nets. Process algebra proved to be a very useful framework for developing a clear understanding of the important concepts in the formalization of inheritance of dynamic behavior. The Petri-net framework provides a formalization that is much closer to existing object-oriented methods. It would have been difficult to achieve the two goals of a clear conceptual understanding and a practical framework in a single theory.

Future work. The most important future challenge is to integrate the concept of life-cycle inheritance in an existing object-oriented design method. If OMT, OOD, or UML is chosen for this purpose, one could choose to translate the notion of life-cycle inheritance to state-transition diagrams or one could choose to replace state-transition diagrams by Petri nets. It is also possible to integrate life-cycle inheritance in an object-oriented formalism based on Petri nets, such as OPN (LAKOS, 1995a). An advantage of this approach is that it results in an integrated framework with a sound theoretical basis. A disadvantage is that object-oriented languages based on Petri nets are not yet in common use. An advantage of incorporating the results of this chapter in a framework as OMT, OOD, or UML is that it will be easier to get acceptance of the notion of life-cycle inheritance in practice, particularly when it is translated to state-transition diagrams.

An aspect that plays an important role in the literature on inheritance is the so-called substitutability principle of (WEGNER AND ZDONIK, 1988). The substitutability principle says that an object of some subclass can always be used in a context where an object of its superclass is expected. When incorporating the inheritance concept developed in this chapter in a complete object-oriented design method, it must be investigated to what extent, or under which assumptions, the four inheritance relations adhere to the substitutability principle. To answer this question, it is not sufficient to consider life-cycle specifications in isolation. It is also necessary to take into account the static structure of classes as well as the interaction between objects.

Another topic for further study is the set of transformation rules to construct subclasses from object life cycles. It is interesting to study variants of the current transformation rules, trying to improve their generality or their efficiency. It would also be useful to translate the transformation rules to state-transition diagrams. Furthermore, it is interesting to transform the rules into transformation rules that preserve liveness and boundedness of ordinary P/T nets. One application of such transformation rules is that they can be used to construct object life cycles.

Future case studies with life-cycle inheritance might reveal that there is a need for a small extension of the framework. When verifying a subclass relationship between two object life cycles in the current framework, it is not allowed to treat different calls of the same method in a different way. All calls of the same method are either blocked, or hidden, or left untouched. However, it is not difficult to define variants of the four inheritance relations of this chapter that allow a different treatment of different calls of the same method. It simply requires the use of temporary method names to distinguish the different groups of method calls. For example, a variant of protocol inheritance could be defined as follows. If an object life cycle is a subclass under protocol inheritance of another object life cycle, then any renaming of the methods new in the subclass yields a subclass of the life cycle under the variant of protocol inheritance. By choosing an appropriate renaming, it is possible that a method already present in the superclass acts as a guard. Thus, by using temporary names for methods acting as guards in the subclass, it is possible to distinguish new invocations of such a method in the subclass from calls of the same method already present in the object life cycle of the superclass.

Finally, an interesting topic for further study is the application of the concepts developed in this chapter to workflow management. Petri nets are well suited to define and analyze workflow processes (see, for example, VAN DER AALST, 1997, 1998). Object life cycles and P/T nets modeling workflow processes are essentially the same. The concept of life-cycle inheritance appears to be particularly useful in the area of ad hoc workflow management. An important topic in this area of workflow management is the adaptation of workflow processes in such a way that the new process preserves the behavior of the original process. A small example of the use of life-cycle inheritance in workflow management can be found in (VAN DER AALST AND BASTEN, 1997).

Bibliographical remarks. The literature on object-oriented design and its theoretical foundations contains several studies related to the research described in this chapter. In (WIERINGA, 1990), abstraction in a process-algebraic setting is suggested as an inheritance relation for dynamic behavior. However, only a few examples are studied. Based on these examples, the author concludes that abstraction is useful but not always sufficient to capture desirable subclass relationships. Life-cycle inheritance as introduced in this thesis, which combines abstraction with encapsulation, is sufficiently powerful to prove the desired subclass relationships in the examples of (WIERINGA, 1990).

Other research on inheritance of dynamic behavior or related concepts such as subtyping describes both formal studies, such as (AMERICA, 1990; EBERT AND ENGELS, 1994; KAPPEL AND SCHREFL, 1995; NIERSTRASZ, 1993; PUNTIGAM, 1996), and practical studies, such as (LAKOS, 1995b; MCGREGOR AND DYER, 1993). The formal studies all take a specific formalism for specifying the dynamic behavior of a class as a starting point; the practical studies start from concrete examples of desired subclass relationships. The result is a wide variety of inheritance and subtype relations. No two of these relations are identical. However, in many cases, the relations show similarities to either protocol inheritance or projection inheritance, although the corresponding concepts of blocking and hiding method calls are never mentioned.

The variety in inheritance relations is not very surprising if one considers, for example, the large number of semantics that exist for concurrent systems (see, for example, VAN GLABBEEK, 1993). There simply is no single semantics of concurrent systems that is suitable for all purposes. Similarly, it cannot be expected that there is a single inheritance relation for dynamic behavior that is always useful. However, it is promising that similar concepts appear to play a role in many of the inheritance and subtype relations for dynamic behavior currently described in the literature.

The research presented in this chapter is also related to some work in the area of concurrency theory, namely conformance testing (BRINKSMA ET AL., 1987). Conformance testing is the problem of verifying by means of testing whether some system implementation satisfies a given specification. In (BRINKSMA ET AL., 1987), a testing relation called "extends" is described, which is related to protocol inheritance.

Contribution. The main contribution of this chapter is the complete formalization of the concept of inheritance of dynamic behavior, starting from the intuitive notions of hiding and blocking method calls. In two different formalisms, four inheritance relations are defined and transformation rules are presented that can be used to construct subclasses from some given class. To validate the results, the theory is applied to a small case study. The fact that the concepts presented in this chapter transcend the two formalisms of process algebra and Petri nets supports the conclusion that the notions of hiding and blocking method calls play an important role in studying and formalizing inheritance of dynamic behavior.

5

Concluding Remarks

Petri nets and process algebra are two different formalisms for specifying and analyzing concurrent systems. In this thesis, two topics combining Petri nets and process algebra have been studied. As explained in Chapter 1, the aim of these studies is twofold. First, they should contribute to a better understanding of the strengths and weaknesses of both Petri nets and process algebra. Second, the studies should stimulate the acceptance of formal methods in design practice and, preferably, lead to improved methods for system design. This chapter tries to answer the question to what extent each of these goals has been achieved.

Chapter 3 has presented a simple design method supporting compositional design by means of Petri nets and process algebra. The method focuses on the correctness of the communication behavior between system components and between the system and its environment. It distinguishes four activities, namely behavioral specification, Petri-net modeling, simulation, and algebraic verification. The theoretical foundation of the method is formed by so-called modular P/T nets and their semantics in a process algebra specifically developed for this purpose. Two case studies have been performed to validate the method in practical design situations.

The method and the underlying theory clearly exploit the strengths of both Petri nets and process algebra. Thus, the method could, in principle, be a useful aid in designing complex concurrent systems. However, the two case studies show that the method is not yet mature. At least two extensions are necessary, namely efficient tool support and the integration of data in the underlying theory.

Chapter 4 has studied the concept of inheritance of dynamic behavior. Current object-oriented design methods lack a precise definition of this concept. In a simple process-algebraic setting, it is shown that the notions of blocking and hiding method calls play an important role in formalizing inheritance of dynamic behavior. Blocking method calls corresponds to the algebraic concepts of encapsulation, whereas hiding method calls corresponds to abstraction. In a process-algebraic setting, these two concepts are well known and have been investigated in detail. However, the algebraic framework is not sufficiently powerful for practical applications. Therefore, the results obtained in the algebraic setting are translated to a framework based on Petri nets. The Petri-net framework resembles formalisms for specifying the behavior of objects which are used in several object-oriented design methods common in current design practice. A small case study has been performed to validate the chosen approach to inheritance of dynamic behavior.

Chapter 4 shows that it can be useful and inspiring to study a single concept in two different settings. The results suggest that the approach to inheritance of dynamic behavior is quite universal and carries over to other formalisms. However, Chapter 4 does not yet integrate the concept of inheritance of dynamic behavior into a full-fledged object-oriented design method. Such an integration is necessary to evaluate the usefulness of the concept in real practical design situations.

So, to what extent fulfills this thesis the abovementioned two goals? The two studies in Chapters 3 and 4 show that it is possible and meaningful to combine Petri nets and process algebra exploiting their strengths and overcoming their individual weaknesses. The Petri-net formalism is particularly useful for modeling and analyzing the state space of a concurrent system in combination with its behavior. Process algebra is well suited for problems in which a behavioral analysis and comparison of concurrent systems is more important than the analysis of their states. Thus, it may be concluded that this thesis contributes to a better understanding of the strengths and weaknesses of Petri nets and process algebra. Perhaps the most important lesson to be learned is that it is very important to choose the appropriate formalism when faced with a certain problem.

Unfortunately, the conclusion concerning the second goal mentioned above is not so clear. As explained, none of the results presented in this thesis is immediately applicable in design practice. However, the concepts developed in Chapters 3 and 4 appear to be useful and meaningful. Thus, they might inspire future developments that improve current methods for system design and contribute to the acceptance of formal methods in practice. This thesis fulfills the most important prerequisite for their practical applicability, namely a sound theoretical foundation.

A

List of Axioms

The following table shows an overview of the algebraic axioms used in this thesis. They are given in alphabetical order, using their names as the key for sorting. The first column contains the axiom; the second column gives its name; the third column provides a reference to the table in which the axiom is defined; the last column gives a reference to the page containing this table. For details about the axioms, operators, and variables, the reader is referred to the original definitions in the main part of this thesis.

$x + y = y + x$	$A1$	2.4.1	34
$(x + y) + z = x + (y + z)$	$A2$	2.4.1	34
$x + x = x$	$A3$	2.4.1	34
$(x + y) \cdot z = x \cdot z + y \cdot z$	$A4$	2.4.1	34
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	$A5$	2.4.1	34
$x + \delta = x$	$A6$	2.4.12	38
$\delta \cdot x = \delta$	$A7$	2.4.12	38
$x \parallel y = y \parallel x$	$ASC2$	2.4.30	45
$(x \mathbin{⌊\!⌊} y) \mathbin{⌊\!⌊} z = x \mathbin{⌊\!⌊} (y \parallel z)$	$ASC4$	2.4.30	45
$(x \parallel y) \parallel z = x \parallel (y \parallel z)$	$ASC6$	2.4.30	45
$x \cdot \tau = x$	$B1$	2.4.37	48
$x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z)$	$B2$	2.4.37	48
$x^* y = x \cdot (x^* y) + y$	$BKS1$	2.4.47	52
$x^* (y \cdot z) = (x^* y) \cdot z$	$BKS2$	2.4.47	52
$x^* (y \cdot ((x + y)^* z) + z) = (x + y)^* z$	$BKS3$	2.4.47	52
$\rho_f(x^* y) = \rho_f(x)^* \rho_f(y)$	$BKS4$	2.4.47	52

continued on next page

continued from previous page

$\lambda^I_s(\delta) = \delta$	*CSO*1	3.4.1	66
$\boldsymbol{ca} \upharpoonright I \leq s \Rightarrow \lambda^I_s(a) = a$	*CSO*2	3.4.1	66
$\boldsymbol{ca} \upharpoonright I \not\leq s \Rightarrow \lambda^I_s(a) = \delta$	*CSO*3	3.4.1	66
$\lambda^I_s(a \cdot x) = \lambda^I_s(a) \cdot \lambda^I_{s-\boldsymbol{ca}\upharpoonright I \uplus \boldsymbol{pa}\upharpoonright I}(x)$	*CSO*4	3.4.1	66
$\lambda^I_s(x + y) = \lambda^I_s(x) + \lambda^I_s(y)$	*CSO*5	3.4.1	66
$a \notin H \Rightarrow \partial_H(a) = a$	*D*1	2.4.23	43
$a \in H \Rightarrow \partial_H(a) = \delta$	*D*2	2.4.23	43
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	*D*3	2.4.23	43
$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$	*D*4	2.4.23	43
$\partial_H(x \parallel y) = \partial_H(x) \parallel \partial_H(y)$	*DM*1	2.4.23	43
$\partial_H(x \mathbin{\lfloor\!\lfloor} y) = \partial_H(x) \mathbin{\lfloor\!\lfloor} \partial_H(y)$	*DM*2	2.4.23	43
$\tau^* x = x + \tau \cdot x$	*FIR*	2.4.54	54
$x \parallel y = x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x$	*M*1	2.4.19	40
$a \mathbin{\lfloor\!\lfloor} x = a \cdot x$	*M*2	2.4.19	40
$a \cdot x \mathbin{\lfloor\!\lfloor} y = a \cdot (x \parallel y)$	*M*3	2.4.19	40
$(x + y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z$	*M*4	2.4.19	40
$\rho_f(a) = \boldsymbol{f}(a)$	*RN*1	2.4.19	40
$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$	*RN*2	2.4.19	40
$\rho_f(x \cdot y) = \rho_f(x) \cdot \rho_f(y)$	*RN*3	2.4.19	40
$\rho_f(x \parallel y) = \rho_f(x) \parallel \rho_f(y)$	*RNM*1	2.4.22	42
$\rho_f(x \mathbin{\lfloor\!\lfloor} y) = \rho_f(x) \mathbin{\lfloor\!\lfloor} \rho_f(y)$	*RNM*2	2.4.22	42
$\dfrac{x = y \cdot x + z, \quad y \text{ is a guard}}{x = y^* z}$	*RSP**	2.4.57	55
$\tau_I(\delta) = \delta$	*TA*1	3.4.11	72
$\tau_I(a) = (\boldsymbol{ca} \upharpoonright (U \backslash I), \boldsymbol{pa} \upharpoonright (U \backslash I))$	*TA*2	3.4.11	72
$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	*TA*3	3.4.11	72
$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$	*TA*4	3.4.11	72
$\tau_I(x^* y) = \tau_I(x)^* \tau_I(y)$	*TA*5	3.4.11	72
$\tau_I(x \parallel y) = \tau_I(x) \parallel \tau_I(y)$	*TAM*1	3.4.11	72
$\tau_I(x \mathbin{\lfloor\!\lfloor} y) = \tau_I(x) \mathbin{\lfloor\!\lfloor} \tau_I(y)$	*TAM*2	3.4.11	72
$a \notin I \Rightarrow \tau_I(a) = a$	*TI*1	2.4.40	49
$a \in I \Rightarrow \tau_I(a) = \tau$	*TI*2	2.4.40	49
$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	*TI*3	2.4.40	49
$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$	*TI*4	2.4.40	49
$\tau_I(x \parallel y) = \tau_I(x) \parallel \tau_I(y)$	*TIM*1	2.4.40	49
$\tau_I(x \mathbin{\lfloor\!\lfloor} y) = \tau_I(x) \mathbin{\lfloor\!\lfloor} \tau_I(y)$	*TIM*2	2.4.40	49

Bibliography

AALST, W.M.P. VAN DER (1996). *Structural Characterizations of Sound Workflow Nets*. Computing Science Report 96/23, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands.

AALST, W.M.P. VAN DER (1997). Verification of Workflow Nets. In: P. Azéma and G. Balbo (eds.), *Application and Theory of Petri Nets 1997, 18th. International Conference, ICATPN'97, Proceedings*, Vol. 1248 of *Lecture Notes in Computer Science*, pp. 407–426, Toulouse, France, June 1997. Springer–Verlag, Berlin, Germany.

AALST, W.M.P. VAN DER (1998). The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems, and Computers*, **8**(1): 21–66.

AALST, W.M.P. VAN DER, AND T. BASTEN (1997). Life-Cycle Inheritance: A Petri-Net-Based Approach. In: P. Azéma and G. Balbo (eds.), *Application and Theory of Petri Nets 1997, 18th. International Conference, ICATPN'97, Proceedings*, Vol. 1248 of *Lecture Notes in Computer Science*, pp. 62–81, Toulouse, France, June 1997. Springer–Verlag, Berlin, Germany.

AMERICA, P. (1990). Designing an Object-Oriented Language with Behavioural Subtyping. In: J.W. de Bakker, W.P. de Roever, and G. Rozenberg (eds.), *Foundations of Object-Oriented Languages, REX School/Workshop, Proceedings*, Vol. 489 of *Lecture Notes in Computer Science*, pp. 60–90, Noordwijkerhout, The Netherlands, May/June 1990. Springer–Verlag, Berlin, Germany.

ASPT (1994). *ExSpect Reference Manual,* Release 5.0. ASPT Foundation/Bakkenist Management Consultants. Information available at `http://www.bakkenist.nl/it/exspect/index.htm`.

BAETEN, J.C.M. AND J.A. BERGSTRA (1993). Non Interleaving Process Algebra. In: E. Best (ed.), *CONCUR '93, 4th. International Conference on Concurrency Theory, Proceedings*, Vol. 715 of *Lecture Notes in Computer Science*, pp. 308–323, Hildesheim, Germany, August 1993. Springer–Verlag, Berlin, Germany.

BAETEN, J.C.M., J.A. BERGSTRA, AND J.W. KLOP (1987). Conditional Axioms and α/β-calculus in Process Algebra. In: M. Wirsing (ed.), *Formal Description of Programming Concepts - III, Proceedings of the IFIP TC2/WG2.2 Working Conference*, pp. 53–75, Ebberup, Denmark, August 1986. North–Holland, Amsterdam, The Netherlands.

BAETEN, J.C.M. AND C. VERHOEF (1995). Concrete Process Algebra. In: S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum (eds.), *Handbook of Logic in Computer Sci-*

ence, Vol. 4, *Semantic Modelling*, pp. 149–268. Oxford University Press, Oxford, UK.

BAETEN, J.C.M. AND W.P. WEIJLAND (1990). *Process Algebra*, Vol. 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK.

BASTEN, T. (1996). Branching Bisimilarity is an Equivalence indeed! *Information Processing Letters*, **58**(3): 141–147.

BASTEN, T. AND W.M.P. VAN DER AALST (1996). *A Process-Algebraic Approach to Life-Cycle Inheritance: Inheritance = Encapsulation + Abstraction*. Computing Science Report 96/05, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands.

BASTEN, T. AND W.M.P. VAN DER AALST (1998). Inheritance of Dynamic Behavior: Development of a Groupware Editor. In: *Concurrent Object-Oriented Programming and Petri Nets*, Advances in Petri Nets, Lecture Notes in Computer Science. Springer–Verlag, Berlin, Germany. To appear.

BASTEN, T., R.N. BOL, AND M. VOORHOEVE (1995). Simulating and Analyzing Railway Interlockings in ExSpect. *IEEE Parallel & Distributed Technology, Systems & Applications*, **3**(3): 50–62.

BASTEN, T. AND J. HOOMAN (1998). *Process Algebra in PVS*. Computing Science Report 98/10, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands.

BASTEN, T. AND M. VOORHOEVE (1995a). *An Algebraic Semantics for Hierarchical P/T Nets*. Computing Science Report 95/35, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands. An extended abstract appeared as (BASTEN AND VOORHOEVE, 1995b).

BASTEN, T. AND M. VOORHOEVE (1995b). An Algebraic Semantics for Hierarchical P/T Nets (extended abstract). In: G. De Michelis and M. Diaz (eds.), *Application and Theory of Petri Nets 1995, 16th. International Conference, Proceedings*, Vol. 935 of *Lecture Notes in Computer Science*, pp. 45–65, Torino, Italy, June 1995. Springer–Verlag, Berlin, Germany.

BERGSTRA, J.A., I. BETHKE, AND A. PONSE (1994). Process Algebra with Iteration and Nesting. *The Computer Journal*, **37**(4): 241–258.

BERGSTRA, J.A. AND J.W. KLOP (1982). *Fixed Point Semantics in Process Algebras*. Report IW 206/82, Mathematical Centre, Amsterdam, The Netherlands. A revised version appeared as (BERGSTRA AND KLOP, 1992).

BERGSTRA, J.A. AND J.W. KLOP (1984). Process Algebra for Synchronous Communication. *Information and Control*, **60**(1–3): 109–137.

BERGSTRA, J.A. AND J.W. KLOP (1992). A Convergence Theorem in Process Algebra. In: J.W. de Bakker and J.J.M.M. Rutten (eds.), *Ten Years of Concurrency Semantics*, pp. 164–195. World Scientific, Singapore.

BERTHELOT, G. (1987). Transformations and Decompositions of Nets. In: W. Brauer, W. Reisig, and G. Rozenberg (eds.), *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course*, Vol. 254 of *Lecture Notes in Computer Science*, pp. 359–376, Bad Honnef, Germany, September 1986. Springer–Verlag, Berlin, Germany.

BEST, E., R. DEVILLERS, AND J.G. HALL (1992). The Box Calculus: A New Causal Algebra with Multi-label Communication. In: G. Rozenberg (ed.), *Advances in Petri Nets 1992*, Vol. 609 of *Lecture Notes in Computer Science*, pp. 21–69. Springer–Verlag, Berlin, Germany.

BEST, E. AND C. FERNÁNDEZ C. (1988). *Nonsequential Processes: A Petri Net View*, Vol. 13 of *EATCS monographs on Theoretical Computer Science*. Springer–Verlag, Berlin, Germany.

BEST, E. AND B. GRAHLMANN (1995). *PEP: Documentation and User Guide*. Universität Hildesheim, Institut für Informatik. Information available at `http://theoretica.informatik.uni-oldenburg.de/~pep`.

BOOCH, G. (1994). *Object-Oriented Analysis and Design: With Applications*. Benjamin/Cummings, Redwood City, California, USA.

BOUDOL, G., G. ROUCAIROL, AND R. DE SIMONE (1985). Petri Nets and Algebraic Calculi of Processes. In: G. Rozenberg (ed.), *Advances in Petri Nets 1985*, Vol. 222 of *Lecture Notes in Computer Science*, pp. 41–58. Springer–Verlag, Berlin, Germany.

BOWEN, J.P. (1998). Oxford Formal-Methods Archive. `http://www.comlab.ox.ac.uk/archive/formal-methods.html`.

BRINKSMA, E., G. SCOLLO, AND C. STEENBERGEN (1987). Lotos Specifications, their Implementations, and their Tests. In: *Protocol Specification, Testing and Verification, VI, Proceedings of the IFIP WG 6.1 6th. international workshop*, pp. 349–360, Montreal, Quebec, Canada, June 1986. Elsevier, North-Holland.

COLOM, J.M. AND M. SILVA (1990). Improving the Linearly Based Characterization of P/T Nets. In: G. Rozenberg (ed.), *Advances in Petri Nets 1990*, Vol. 483 of *Lecture Notes in Computer Science*, pp. 113–145. Springer–Verlag, Berlin, Germany.

DEGANO, P., R. DE NICOLA, AND U. MONTANARI (1988). A Distributed Operational Semantics for CCS Based on Condition/Event Systems. *Acta Informatica*, **26**(1/2): 59–91.

DESEL, J. (1990). Reduction and Design of Well-behaved Concurrent Systems. In: J.C.M. Baeten and J.W. Klop (eds.), *CONCUR '90, Theories of Concurrency: Unification and Extension, Proceedings*, Vol. 458 of *Lecture Notes in Computer Science*, pp. 166–181, Amsterdam, The Netherlands, August 1990. Springer–Verlag, Berlin, Germany.

DESEL, J. AND J. ESPARZA (1995). *Free Choice Petri Nets*, Vol. 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK.

DIETZ, C. AND G. SCHREIBER (1994). A Term Representation of P/T Systems. In: R. Valette (ed.), *Application and Theory of Petri Nets 1994, 15th. International Conference, Proceedings*, Vol. 815 of *Lecture Notes in Computer Science*, pp. 239–257, Zaragoza, Spain, June 1994. Springer–Verlag, Berlin, Germany.

DILLON, L.K. AND S. SANKAR (1997). Introduction to the Special Issue. *IEEE Transactions on Software Engineering, Special Issue on Formal Methods in Software Practice*, **23**(5): 265–266.

EBERT, J. AND G. ENGELS (1994). Structural and Behavioural Views on OMT-Classes. In: E. Bertino and S. Urban (eds.), *Object-Oriented Methodologies and Systems, International Symposium, ISOOMS'94, Proceedings*, Vol. 858 of *Lecture Notes in Computer Science*, pp. 142–157, Palermo, Italy, September 1994. Springer–Verlag, Berlin, Germany.

ESPARZA, J. (1990). Synthesis Rules for Petri Nets, and How they Lead to New Results. In: J.C.M. Baeten and J.W. Klop (eds.), *CONCUR '90, Theories of Concurrency: Unification and Extension, Proceedings*, Vol. 458 of *Lecture Notes in Computer Science*, pp. 182–198, Amsterdam, The Netherlands, August 1990. Springer–Verlag, Berlin, Germany.

ESPARZA, J. AND M. NIELSEN (1994). Decibility Issues for Petri Nets - a survey. *Journal of Information Processing and Cybernetics*, **30**(3): 143–160.

FOKKINK, W.J. AND H. ZANTEMA (1994). Basic Process Algebra with Iteration: Completeness of its Equational Axioms. *The Computer Journal*, **37**(4): 259–267.

FOWLER, M. AND K. SCOTT (1997). *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, Massachusetts, USA.

GLABBEEK, R.J. VAN (1990a). *Comparative Concurrency Semantics and Refinement of Actions*. Ph.D. thesis, Free University, Amsterdam, The Netherlands.

GLABBEEK, R.J. VAN (1990b). The Linear Time – Branching Time Spectrum. In: J.C.M. Baeten and J.W. Klop (eds.), *CONCUR '90, Theories of Concurrency: Unification and Extension, Proceedings*, Vol. 458 of *Lecture Notes in Computer Science*, pp. 278–297, Amsterdam, The Netherlands, August 1990. Springer–Verlag, Berlin, Germany.

GLABBEEK, R.J. VAN (1993). The Linear Time – Branching Time Spectrum II: The Semantics of Sequential Systems with Silent Moves (extended abstract). In: E. Best (ed.), *CONCUR '93, 4th. International Conference on Concurrency Theory, Proceedings*, Vol. 715 of *Lecture Notes in Computer Science*, pp. 66–81, Hildesheim, Germany, August 1993. Springer–Verlag, Berlin, Germany.

GLABBEEK, R.J. VAN (1994). What is Branching Time Semantics and Why to Use It? In: *Bulletin of the EATCS,* No. 53, pp. 191–198. European Association for Theoretical Computer Science.

GLABBEEK, R.J. VAN (1997). Axiomatizing Flat Iteration. In: A. Mazurkiewicz and J. Winkowski (eds.), *CONCUR '97: Concurrency Theory, 8th. International Con-*

ference, Proceedings, Vol. 1243 of *Lecture Notes in Computer Science*, pp. 228–242, Warsaw, Poland, July 1997. Springer–Verlag, Berlin, Germany.

GLABBEEK, R.J. VAN, AND F.W. VAANDRAGER (1987). Petri Net Models for Algebraic Theories of Concurrency. In: J.W. de Bakker, A.J. Nijman, and P.C. Treleaven (eds.), *PARLE Parallel Architectures and Languages Europe,* Vol. I*, Parallel Architectures*, Vol. 258 of *Lecture Notes in Computer Science*, pp. 224–242, Eindhoven, The Netherlands, June 1987. Springer–Verlag, Berlin, Germany.

GLABBEEK, R.J. VAN, AND W.P. WEIJLAND (1989). Branching Time and Abstraction in Bisimulation Semantics (extended abstract). In: G.X. Ritter (ed.), *Information Processing 89: Proceedings of the IFIP 11th. World Computer Congress*, pp. 613–618, San Francisco, California, USA, August/September 1989. Elsevier Science Publishers B.V., North-Holland. Full version appeared as (VAN GLABBEEK AND WEIJLAND, 1996).

GLABBEEK, R.J. VAN, AND W.P. WEIJLAND (1996). Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, **43**(3): 555–600.

GOLTZ, U. (1988). On Representing CCS Programs by Finite Petri Nets. In: M.P. Chytil, L. Janiga, and V. Koubek (eds.), *Mathematical Foundations of Computer Science 1988, Proceedings*, Vol. 324 of *Lecture Notes in Computer Science*, pp. 339–350, Carlsbad, Czechoslovakia, August/September, 1988. Springer–Verlag, Berlin, Germany.

GRAHLMANN, B. (1997). The PEP Tool. In: O. Grumberg (ed.), *Computer Aided Verification, 9th. International Conference, CAV'97,Proceedings*, Vol. 1254 of *Lecture Notes in Computer Science*, pp. 440–443, Haifa, Israel, June 1997. Springer–Verlag, Berlin, Germany.

GRAHLMANN, B. AND E. BEST (1996). PEP: More than a Petri Net Tool. In: T. Margaria and B. Steffen (eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 2nd. International Workshop, TACAS'96, Proceedings*, Vol. 1055 of *Lecture Notes in Computer Science*, pp. 397–401, Passau, Germany, March 1996. Springer–Verlag, Berlin, Germany.

GROOTE, J.F. AND A. PONSE (1994). Proof Theory for μCRL: A Language for Processes with Data. In: D.J. Andrews, J.F. Groote, and C.A. Middelburg (eds.), *Semantics of Specification Languages, Proceedings of the International Workshop SoSL*, Workshops in Computing, pp. 232–251, Utrecht, The Netherlands, October 1993. Springer–Verlag, Berlin, Germany.

GROOTE, J.F. AND F.W. VAANDRAGER (1990). An Efficient Algorithm for Branching and Stuttering Equivalence. In: M.S. Paterson (ed.), *Automata, Languages and Programming, 17th. International Colloquium, Proceedings*, Vol. 443 of *Lecture Notes in Computer Science*, pp. 626–638, Warwick University, England, July 1990. Springer–Verlag, Berlin, Germany.

HEE, K.M. VAN (1994). *Information Systems Engineering: A Formal Approach.* Cambridge University Press, Cambridge, UK.

HEE, K.M. VAN, L.J. SOMERS, AND M. VOORHOEVE (1989). Executable Specifications for Distributed Information Systems. In: E.D. Falkenberg and P. Lindgreen (eds.), *Information System Concepts: An In-depth Analysis, Proceedings of the IFIP TC8/WG8.1 Working Conference*, pp. 139–156, Namur, Belgium, October 1989. North–Holland, Amsterdam, The Netherlands.

HOARE, C.A.R. (1985). *Communicating Sequential Processes*. Prentice–Hall International, London, UK.

JENSEN, K. (1992). *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use,* Vol. 1*, Basic Concepts*. EATCS monographs on Theoretical Computer Science. Springer–Verlag, Berlin, Germany.

JENSEN, K. (1995). *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use,* Vol. 2*, Analysis Methods*. Monographs in Theoretical Computer Science: An EATCS Series. Springer–Verlag, Berlin, Germany.

JENSEN, K. (1997). *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use,* Vol. 3*, Practical Use*. Monographs in Theoretical Computer Science: An EATCS Series. Springer–Verlag, Berlin, Germany.

JENSEN, K., S. CHRISTENSEN, P. HUBER, AND M. HOLLA (1991). *Design/CPN: A Reference Manual*. Meta Software Corporation. Information available at `http://www.daimi.au.dk/designCPN`.

KAPPEL, G. AND M. SCHREFL (1995). Inheritance of Object Behavior - Consistent Extension of Object Life Cycles. In: J. Eder and L.A. Kalinichenko (eds.), *East/West Database Workshop, Proceedings of the 2nd. International Workshop*, Workshops in Computing, pp. 289–300, Klagenfurt, Austria, September 1994. Springer–Verlag, Berlin, Germany.

KLEENE, S.C. (1956). Representation of Events in Nerve Nets and Finite Automata. In: C.E. Shannon and J. McCarthy (eds.), *Automata Studies*, No. 34 in Annals of Mathematics Studies, pp. 3–41. Princeton University Press, Princeton, New Jersey, USA.

KOVALYOV, A.V. (1990). On Complete Reducibility of Some Classes of Petri Nets. In: *Application and Theory of Petri Nets 1990, 11th. International Conference, ICATPN'90, Proceedings*, pp. 352–366, Paris, France, June 1990.

KOVALYOV, A.V. (1996). An $O(|S| \times |T|)$-Algorithm to Verify if a Net is Regular. In: J. Billington and W. Reisig (eds.), *Application and Theory of Petri Nets 1996, 17th. International Conference, ICATPN'96, Proceedings*, Vol. 1091 of *Lecture Notes in Computer Science*, pp. 366–379, Osaka, Japan, June 1996. Springer–Verlag, Berlin, Germany.

LAKOS, C.A. (1995a). From Coloured Petri Nets to Object Petri Nets. In: G. De Michelis and M. Diaz (eds.), *Application and Theory of Petri Nets 1995, 16th. International Conference, Proceedings*, Vol. 935 of *Lecture Notes in Computer Science*, pp. 278–297, Torino, Italy, June 1995. Springer–Verlag, Berlin, Germany.

LAKOS, C.A. (1995b). Pragmatic Inheritance Issues for Object Petri Nets. In: *TOOLS Pacific 1995, Proceedings*, pp. 309–321, Melbourne, Australia. Prentice-Hall.

MCGREGOR, J.D. AND D.M. DYER (1993). A Note on Inheritance and State Machines. *Software Engineering Notes*, **18**(4): 1–15.

MILNER, R. (1980). *A Calculus of Communicating Systems*, Vol. 92 of *Lecture Notes in Computer Science*. Springer–Verlag, Berlin, Germany.

MILNER, R. (1989). *Communication and Concurrency*. Prentice–Hall International, London, UK.

MONTANARI, U. AND D. YANKELEVICH (1994). Combining CCS and Petri Nets via Structural Axioms. *Fundamenta Informaticae*, **20**(1–3): 193–229.

MURATA, T. (1989). Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, **77**(4): 541–580.

NIERSTRASZ, O. (1993). Regular Types for Active Objects. *ACM Sigplan Notices*, **28**(10): 1–15. Proceedings of the 8th. annual conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'93, Washington DC, 1993.

OLDEROG, E.-R. (1987). Petri Nets and Algebraic Calculi of Processes. In: G. Rozenberg (ed.), *Advances in Petri Nets 1987*, Vol. 266 of *Lecture Notes in Computer Science*, pp. 196–223. Springer–Verlag, Berlin, Germany.

OWRE, S., J. RUSHBY, N. SHANKAR, AND F. VON HENKE (1995). Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, **21**(2): 107–125.

PARK, D. (1981). Concurrency and Automata on Infinite Sequences. In: P. Deussen (ed.), *Theoretical Computer Science, 5th. GI-Conference*, Vol. 104 of *Lecture Notes in Computer Science*, pp. 167–183, Karlsruhe, Germany, March 1981. Springer–Verlag, Berlin, Germany.

PETERSON, J.L. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, USA.

PETRI, C.A. (1962). *Kommunikation mit Automaten*. Ph.D. thesis, Institut für instrumentelle Mathematik, Bonn, Germany. In German.

PUNTIGAM, F. (1996). Types for Active Objects Based on Trace Semantics. In: *Formal Methods for Open Object-based Distributed Systems, 1st. IFIP international workshop, Proceedings*, Paris, France, March 1996.

RAMBAGS, P.M.P. (1994). *Decomposition and Protocols in High-level Petri Nets*. Ph.D. thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands.

REISIG, W. (1985). *Petri Nets: An Introduction*, Vol. 4 of *EATCS monographs on Theoretical Computer Science*. Springer–Verlag, Berlin, Germany.

RENSINK, A. AND H. WEHRHEIM (1994). Weak Sequential Composition in Process Algebras. In: B. Jonsson and J. Parrow (eds.), *CONCUR '94: Concurrency Theory, 5th. International Conference, Proceedings*, Vol. 836 of *Lecture Notes in Computer*

Science, pp. 226–241, Uppsala, Sweden, August 1994. Springer–Verlag, Berlin, Germany.

RUMBAUGH, J., M. BLAHA, W. PREMERLANI, F. EDDY, AND W. LORENSEN (1991). *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, USA.

SEWELL, P. (1994). Bisimulation is not Finitely (First Order) Equationally Axiomatisable. In: *1994 IEEE 9th. Annual Symposium on Logic in Computer Science, LICS '94, Proceedings*, pp. 62–70, Paris, France, July 1994. IEEE Computer Society Press, Los Alamitos, California, USA.

TAUBNER, D. (1989). *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets*, Vol. 369 of *Lecture Notes in Computer Science*. Springer–Verlag, Berlin, Germany.

VALETTE, R. (1979). Analysis of Petri Nets by Stepwise Refinements. *Journal of Computer and System Sciences*, **18**(1): 35–46.

WEGNER, P. AND S.B. ZDONIK (1988). Inheritance as an Incremental Modification Mechanism or What like is and isn't like. In: S. Gjessing and K. Nygaard (eds.), *ECOOP '88, European Conference on Object-Oriented Programming, Proceedings*, Vol. 322 of *Lecture Notes in Computer Science*, pp. 55–77, Oslo, Norway, August 1988. Springer–Verlag, Berlin, Germany.

WEIJLAND, W.P. (1989). *Synchrony and Asynchrony in Process Algebra*. Ph.D. thesis, University of Amsterdam, Department of Mathematics and Computer Science, Amsterdam, The Netherlands.

WIERINGA, R.J. (1990). *Algebraic Foundations for Dynamic Conceptual Models*. Ph.D. thesis, Free University, Amsterdam, The Netherlands.

Index

Boldface page numbers refer to the definition of mathematical concepts, as well as the introduction of terminology or notation. Page numbers in normal font refer to important information on a subject, illustrative examples, or the informal explanation of concepts.

H

I

K

L

M

N

O

P

R

U

V

W

Summary

To date, there are many different formal methods for describing and analyzing concurrent systems. Two of the most widely used formalisms are Petri nets and process algebra. The main motivation of this thesis is to study topics combining these two formalisms. The first goal is to learn more about the strengths and the weaknesses of both Petri nets and process algebra. The second goal is to stimulate the use and acceptance of formal methods in design practice. The thesis can be divided into three parts.

The first part gives an introduction to basic concurrency theory. The starting point is a framework of labeled transition systems which can be used to precisely define the dynamic behavior of concurrent systems. The framework is used to provide an operational semantics for Petri-net models and process-algebraic specifications of concurrent systems.

Petri nets combine an intuitive graphical representation with powerful analysis techniques. In the first part of this thesis, the class of labeled P/T nets is described to illustrate the fundamentals of Petri-net modeling. A labeled-P/T-net model of a concurrent system describes both the reachable states and the possible state transitions of the system in a natural way.

Process algebra is a purely symbolic formalism which is particularly well suited for computer manipulation. This thesis focuses on ACP-style process algebra. A process-algebraic specification of a concurrent system describes the dynamic behavior of the system in a compact way while abstracting away from the details of its states. The basic analysis technique in ACP-style process algebra is equational reasoning.

The second part of this thesis combines Petri nets and process algebra in a method supporting the compositional design of concurrent systems. The method consists of four activities, namely behavioral specification, Petri-net modeling, simulation, and algebraic verification. The first activity focuses on describing the dynamic behavior of the system under consideration. In the second activity, a complete Petri-net model of the system is created, which usually contains more detail than the behavioral specification. Third, simulation can be used to validate the Petri-net model and the behavioral specification of the system. Finally, algebraic verification is used to prove formally that the Petri-net model satisfies the behavioral specification.

This thesis mainly focuses on the second and fourth activity of the above method. To allow component-based design of concurrent systems, the formalism of modular P/T nets is introduced. The operational semantics of modular P/T nets is formalized in a process algebra specifically developed for this purpose. The theory of this thesis makes it is possi-

ble to verify, by means of equational reasoning, the behavior of a modular P/T net against a behavioral specification.

Two case studies are described to validate the design method and the underlying theory in practical design situations. The main conclusion is that Petri nets and process algebra complement each other very well, but that the method and theory are not yet mature. A further extension of the theory as well as the development of tool support is a prerequisite for successful practical applications.

The third part of this thesis describes an approach to inheritance of dynamic behavior. Most object-oriented design methods currently used in practice contain a more or less precise definition of inheritance for the attributes and methods of classes. However, they lack a definition of inheritance for object life cycles that describe the dynamic behavior of objects.

This thesis formalizes inheritance of dynamic behavior both in a process-algebraic setting and in a Petri-net framework. In the chosen approach, blocking and hiding method calls play a crucial role. In process algebra, these two notions correspond to the well-investigated concepts of encapsulation and abstraction, respectively. Four inheritance relations for dynamic behavior are defined and studied in a simple process algebra. The definitions and results are translated to a framework based on labeled P/T nets, which is a framework much closer to existing object-oriented methods than the algebraic framework. In the P/T-net framework, a set of transformation rules is developed that can be used to construct classes inheriting the life cycle of another class. A small case study shows that these transformation rules are useful in structuring the design process and that they stimulate the reuse of life-cycle specifications.

The main conclusion of the third part of this thesis is that it can be inspiring to study a single topic in two different formalisms. The formalization of inheritance of dynamic behavior in process algebra provides a clear understanding of the concept, whereas the formalization in Petri nets yields a framework close to practice. However, one important challenge remains, namely the integration of the notion of inheritance developed in this thesis in a full-fledged object-oriented design method.

Summarizing, it may be concluded that this thesis contributes to a better understanding of the strengths and weaknesses of Petri nets and process algebra. The concepts developed in this thesis appear to be sound and promising, although it is not yet clear to what extent the results will stimulate the use of formal methods in design practice.

Samenvatting

Vandaag de dag zijn er vele verschillende formele methoden voor het beschrijven en analyseren van coöperatieve systemen[1]. Twee van de meest gebruikte formalismen zijn Petri netten en procesalgebra. De bestudering van onderwerpen die beide formalismen combineren is de belangrijkste motivatie van dit proefschrift. De eerste doelstelling is om meer te leren over de sterke en zwakke punten van zowel Petri netten als procesalgebra. De tweede doelstelling is het gebruik en de acceptatie van formele methoden in de ontwerppraktijk te stimuleren. In dit proefschrift kunnen drie delen onderscheiden worden.

Het eerste deel introduceert elementaire theorie over coöperatieve systemen. Het uitgangspunt is een raamwerk van gelabelde transitie systemen dat dient om het dynamisch gedrag van coöperatieve systemen precies vast te leggen. Het raamwerk wordt gebruikt om Petri-net modellen en procesalgebraïsche specificaties van coöperatieve systemen te voorzien van een operationele semantiek.

Petri netten combineren een intuitïeve grafische representatie met krachtige analyse technieken. Om de fundamentele eigenschappen van Petri-net modellering te illustreren, wordt in het eerste deel van dit proefschrift de klasse van gelabelde P/T netten beschreven. Een gelabeld-P/T-net model van een coöperatief systeem beschrijft op natuurlijke wijze zowel de bereikbare toestanden als de mogelijke toestandsovergangen van het systeem.

Procesalgebra is een volledig symbolisch formalisme dat in het bijzonder geschikt is voor manipulatie door computers. Dit proefschrift richt zich op ACP-stijl procesalgebra. Een procesalgebraïsche specificatie van een coöperatief systeem beschrijft op beknopte wijze het dynamisch gedrag van het systeem, terwijl zij abstraheert van de details van de toestanden van het systeem. De elementaire analyse techniek in ACP-stijl procesalgebra is manipulatie van vergelijkingen.

Het tweede deel van dit proefschrift beschrijft een methode die Petri netten en procesalgebra combineert en die het compositioneel ontwerpen van coöperatieve systemen ondersteunt. De methode bestaat uit vier activiteiten, namelijk specificatie van gedrag, Petri-net modellering, simulatie en algebraïsche verificatie. De eerste activiteit is gericht op het beschrijven van het dynamisch gedrag van het te beschouwen systeem. Gedurende de tweede activiteit wordt een volledig Petri-net model van het system gecreëerd, dat gewoonlijk meer details beschrijft dan de gedragsspecificatie. De derde activiteit bestaat uit simulatie gericht op de validatie van het Petri-net model en de gedragsspecificatie van het systeem. In de vierde activiteit, tenslotte, wordt algebraïsche verificatie toegepast om formeel te bewijzen dat het Petri-net model voldoet aan de gedragsspecificatie.

[1](Ned.) coöperatief systeem = (Eng.) concurrent system

Dit proefschrift is voornamelijk gericht op de tweede en de vierde activiteit van de eerder beschreven methode. Om het modulair ontwerpen van coöperatieve systemen mogelijk te maken, wordt het formalisme van modulaire P/T netten geïntroduceerd. De operationele semantiek van modulaire P/T netten wordt geformaliseerd in een procesalgebra die specifiek voor dit doel ontwikkeld is. De theorie in dit proefschrift maakt het mogelijk om door manipulatie van vergelijkingen het gedrag van een modulair P/T net te verifiëren tegen een gedragsspecificatie.

Twee praktijkstudies worden beschreven om de ontwerpmethode en de onderliggende theorie te toetsen. De belangrijkste conclusie is dat Petri netten en procesalgebra elkaar goed aanvullen, maar dat de methode en de theorie nog niet volwassen zijn. Een verdere uitbreiding van de theorie en de ontwikkeling van ondersteunde gereedschappen is een voorwaarde voor succesvolle toepassing in de praktijk.

Het derde deel van dit proefschrift beschrijft een aanpak betreffende overerving[2] van dynamisch gedrag. De meeste object-georiënteerde ontwerpmethoden die vandaag de dag in de praktijk gebruikt worden, hebben een min of meer exacte definitie van overerving voor de attributen en de methoden van een klasse. Een goede definitie van overerving voor de levenscyclus van een object, die het dynamisch gedrag van objecten van de klasse beschrijft, ontbreekt echter.

Dit proefschrift formaliseert overerving van dynamisch gedrag zowel in een procesalgebraïsche setting als in een raamwerk van Petri netten. Het blokkeren en verbergen van methode-aanroepen speelt een cruciale rol in de gekozen aanpak. In procesalgebra corresponderen deze twee noties met de goed onderzochte concepten van respectievelijk encapsulatie en abstractie. Vier overervingsrelaties voor dynamisch gedrag worden gedefinieerd en bestudeerd in een eenvoudige procesalgebra. De definities en resultaten worden vertaald naar een raamwerk gebaseerd op gelabelde P/T netten, dat veel dichter bij bestaande object-georiënteerde methoden staat dan het algebraïsche raamwerk. In het P/T-net raamwerk wordt een set transformatieregels ontwikkeld die gebruikt kan worden om klassen te construeren die de levenscyclus van een andere klasse overerven. Een kleine praktijkstudie laat zien dat deze transformatieregels gebruikt kunnen worden om het ontwerpproces te structureren en dat zij hergebruik van specificaties van levenscycli stimuleren.

De belangrijkste conclusie van het derde deel van dit proefschrift is dat het inspirerend kan zijn om een enkel onderwerp in twee verschillende formalismen te bestuderen. De formalisering van overerving van dynamisch gedrag in procesalgebra geeft een helder begrip van het concept, terwijl de formalisering in Petri netten een raamwerk oplevert dat dicht bij de praktijk staat. Er blijft echter één belangrijke uitdaging over, namelijk de integratie van de notie van overerving die ontwikkeld is in dit proefschrift, in een volledige object-georiënteerde ontwerpmethode.

Samenvattend mag geconcludeerd worden dat dit proefschrift bijdraagt aan een beter begrip van de sterke en zwakke punten van Petri netten en procesalgebra. De in dit proefschrift ontwikkelde concepten lijken gezond en veelbelovend, alhoewel het nog niet duidelijk is in welke mate de resultaten het gebruik van formele methoden in de ontwerppraktijk zullen stimuleren.

[2](Ned.) overerving = (Eng.) inheritance

Curriculum Vitae

Twan Basten was born on May 28, 1969 in Sevenum, the Netherlands, but he grew up in Eijsden, a little village near Maastricht. From 1981 until 1987, he went to secondary school at the Jeanne d'Arc College in Maastricht.

After he finished his secondary education, he started his studies in Computing Science (Technische Informatica) at the Eindhoven University of Technology in Eindhoven, the Netherlands. During his studies, he organized a three-week study trip to Canada. On the flight back to the Netherlands, he decided that he would return to Canada to work on his Master's thesis. From September 1992 until July 1993, he visited the University of Waterloo, Ontario, Canada, as a research assistant. Under the supervision of Dr. Jay P. Black and Dr. Mike H. Coffin, he worked on event-based debugging of distributed programs. At the end of his stay in Waterloo, he wrote his Master's thesis. In August 1993, he graduated cum laude at the Eindhoven University of Technology under the supervision of Prof.Dr.Dipl.Ing. Dieter K. Hammer.

After working several months as a research assistant (toegevoegd onderzoeker) at the Eindhoven University of Technology, Twan Basten became a Ph.D. student (assistent in opleiding) at the same university in February 1994. Under the supervision of Prof.Dr. Jos C.M. Baeten and Prof.Dr. Kees M. van Hee, he worked four and a half years on various topics related to process algebra and Petri nets. This thesis contains the most important results of this work.

Since August 1998, Twan Basten has been working as an assistant professor (universitair docent) at the Eindhoven University of Technology. His main research area is workflow management, in particular, the formal specification and analysis of workflows by means of Petri nets. Twan Basten can be reached at `tbasten@win.tue.nl`.

Titles in the IPA Dissertation Series

The State Operator in Process Algebra
J. O. Blanco
Faculty of Mathematics and Computing Science, TUE, 1996-1

Transformational Development of Data-Parallel Algorithms
A. M. Geerling
Faculty of Mathematics and Computer Science, KUN, 1996-2

Interactive Functional Programs: Models, Methods, and Implementation
P. M. Achten
Faculty of Mathematics and Computer Science, KUN, 1996-3

Parallel Local Search
M. G. A. Verhoeven
Faculty of Mathematics and Computing Science, TUE, 1996-4

The Implementation of Functional Languages on Parallel Machines with Distrib. Memory
M. H. G. K. Kesseler
Faculty of Mathematics and Computer Science, KUN, 1996-5

Distributed Algorithms for Hard Real-Time Systems
D. Alstein
Faculty of Mathematics and Computing Science, TUE, 1996-6

Communication, Synchronization, and Fault-Tolerance
J. H. Hoepman
Faculty of Mathematics and Computer Science, UvA, 1996-7

Reductivity Arguments and Program Construction
H. Doornbos
Faculty of Mathematics and Computing Science, TUE, 1996-8

Functorial Operational Semantics and its Denotational Dual
D. Turi
Faculty of Mathematics and Computer Science, VUA, 1996-9

Single-Rail Handshake Circuits
A. M. G. Peeters
Faculty of Mathematics and Computing Science, TUE, 1996-10

A Systems Engineering Specification Formalism
N. W. A. Arends
Faculty of Mechanical Engineering, TUE, 1996-11

Normalisation in Lambda Calculus and its Relation to Type Inference
P. Severi de Santiago
Faculty of Mathematics and Computing Science, TUE, 1996-12

Abstract Interpretation and Partition Refinement for Model Checking
D. R. Dams
Faculty of Mathematics and Computing Science, TUE, 1996-13

Topological Dualities in Semantics
M. M. Bonsangue
Faculty of Mathematics and Computer Science, VUA, 1996-14

Algorithms for Graphs of Small Treewidth
B. L. E. de Fluiter
Faculty of Mathematics and Computer Science, UU, 1997-01

Process-algebraic Transformations in Context
W. T. M. Kars
Faculty of Computer Science, UT, 1997-02

A Generic Theory of Data Types
P. F. Hoogendijk
Faculty of Mathematics and Computing Science, TUE, 1997-03

The Evolution of Type Theory in Logic and Mathematics
T. D. L. Laan
Faculty of Mathematics and Computing Science, TUE, 1997-04

Preservation of Termination for Explicit Substitution
C. J. Bloo
Faculty of Mathematics and Computing Science, TUE, 1997-05

Discrete-Time Process Algebra
J. J. Vereijken
Faculty of Mathematics and Computing Science, TUE, 1997-06

A Functional Approach to Syntax and Typing
F. A. M. van den Beuken
Faculty of Mathematics and Computer Science, KUN, 1997-07

Ins and Outs in Refusal Testing
A.W. Heerink
Faculty of Computer Science, UT, 1998-01

A Discrete-Event Simulator for Systems Engineering

G. Naumoski and W. Alberts
Faculty of Mechanical Engineering, TUE, 1998-02

Scheduling with Communication for Multiprocessor Computation
J. Verriet
Faculty of Mathematics and Computer Science, UU, 1998-03

An Asynchronous Low-Power 80C51 Microcontroller
J. S. H. van Gageldonk
Faculty of Mathematics and Computing Science, TUE, 1998-04

In Terms of Nets: System Design with Petri Nets and Process Algebra
A. A. Basten
Faculty of Mathematics and Computing Science, TUE, 1998-05