Behaviour and Refinement of Port-Based Components with Synchronous and Asynchronous Communication

Stephan Janisch

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik der
Ludwig-Maximilians-Universität München
zur Erlangung des Grades Doctor rerum naturalium (Dr. rer. nat.)

Erster Gutachter:	Prof. Dr. Rolf Hennicker, LMU München, IFI, PST
Auswärtiger Gutachter:	Prof. Dr. Stephan Merz, INRIA Lorraine, LORIA
Dritter Gutachter:	Prof. Dr. Martin Wirsing, LMU München, IFI, PST

Abgabedatum:	18. Juni 2010
Tag der mündlichen Prüfung:	27. Juli 2010

Abstract

Component-based development is an established discipline of Software Engineering. It focuses on the development of strongly encapsulated components to support reuse within the construction of hierarchical systems by assemblies of components and their connectors. Software component models define concepts for the construction of component-based systems. Formal software component models additionally define a model of dynamic component behaviour and support formal analysis which is of paramount importance for the verification of behavioural properties and component correctness. This holds in particular for the development of concurrent component-based systems.

Existing formal component models usually use a synchronous rendezvous mechanism to define the global behaviour of communicating concurrent components. However, in order to support the specification and analysis of concurrent systems an asynchronous communication timing, as prominent for example in FIFO buffered communications of message-passing systems, must also be taken into account. Moreover, in order to support component-based development of hierarchical systems with top-down or bottom-up design approaches, it is crucial to distinguish between behaviour specifications and behaviour implementations. Still, an integrated view of specification and implementation of component behaviour must be provided.

In this thesis, we develop a formal software component model with hierarchical port-based components, providing an integrated approach for the static and dynamic modelling of concurrent message-passing systems with synchronous and asynchronous message exchange. We consider components that execute in parallel to each other and communicate solely by message exchange via ports. Binary connectors between ports are utilised for synchronous (rendezvous) or asynchronous (FIFO buffered) message exchange. Components and connectors are used for a compositional construction of assemblies. Assemblies in turn can be used for the implementation of composite components by delegating messages between the ports of the composite component and the open ports of the assembly.

The proposed model distinguishes abstract models for component implementations and component specifications. We call the former “component behaviours” and the latter “component frames”. Behaviours and frames are formally related by a definition of implementation correctness. We show that our model supports top-down and bottom-up design approaches, and component-wise evolution [**dAH01b**] in hierarchical systems. As a formal model of behaviours we use I/O-transition systems (IOTSs) and for frames we use input-persistent I/O-transition systems (PIOs). IOTSs are in fact a special case of PIOs that in turn are a specialisation of modal I/O automata of Larsen and Nyman [**LNW07**].

We develop a theory for refinement and compatibility of PIOs with synchronous and asynchronous communication and show that our refinement relation, called blackbox refinement, satisfies fundamental properties such as transitivity and compositionality. Motivated by the aim to detect communication errors we take into account the asymmetry of sending (output) and receiving (input) messages and develop different notions of output compatibility with varying degrees of freedom for the interleaving of local, non-shared component actions. Based on an encoding of FIFO buffers we show that blackbox refinement is transferable to asynchronously communicating components. Moreover, we develop a notion of asynchronous output compatibility that is a natural extension of synchronous

output compatibility and show its preservation by blackbox refinement. The resulting theory is an interface theory in the sense of [**BMSH10**]. Concerning more general properties we define a greybox variant of blackbox refinement. Greybox refinement allows to distinguish internal labels and we show that greybox refinement preserves safety properties with regard to the communication traces of a composed system.

As a concrete syntax we use UML components with ports for the specification of static structures, state machines for the specification of behaviours, and protocol state machines for frames. By combining an industry-strength modelling language with a formal theory we obtain both a precise and well-understandable specification as well as an unambiguous meaning of dynamic system aspects.

With regard to verification we must take into account that systems of buffered PIOs are potentially infinite-state systems which makes most verification problems undecidable. We consider foremost the verification of asynchronous compatibility and discuss a criterion for closed systems based on synchronous compatibility. For the general verification support of buffered PIO systems we discuss a translation to Communicating Finite State Machines (CFSM) [**vB78**]. We develop a CFSM correspondence of asynchronous compatibility and sketch the application of a symbolic CFSM-based approach to its verification.

Finally, we illustrate the model developed within this thesis, using the Common Component Modelling Example (CoCoME). The CoCoME, initiated as a GI-Dagstuhl research seminar, aimed at a comparison of existing component models using a common requirement specification and reference implementation of a point-of-sale system. We consider UML specifications of static and dynamic system aspects, illustrate their translations to transition systems, and discuss proof obligations arising from behaviour, correctness, and frame analysis.

Acknowledgements

I would like to thank my supervisor Rolf Hennicker for his guidance and support during all the years. The countless discussions about both, concepts and theory, often resulted in new ideas to extend and improve the results of this thesis. Rolf continously insisted on a conceptional sustainable and formally precise development and, by this means, contributed to a component model which is hopefully precise about concepts and, at the same time, in touch with practical applicability.

I am also deeply indebted to Alexander Knapp for many fruitful discussions and the joint work on our publications. Alexander's help in technical, but also in conceptional questions was invaluable. His intuition in formal reasoning repeatedly helped me to move on with the theoretical parts of my research. Moreover, I would like to thank Stephan Merz and Martin Wirsing for their reviews and critical reading of this thesis and the whole PST team for providing a perfect working atmosphere with lots of discussions, especially during the days we spent at our yearly "hut seminar".

Finally, I thank my wife Clarissa da Costa and my whole family for their encouragement and support. I am especially grateful to Gudrun da Costa for her language corrections of the informal parts of this thesis.

Contents

Chapter 1. Introduction 1
1. Component-Based Software Engineering 1
2. Component Models and Distributed Systems 4
3. Contribution and Overview 9

Chapter 2. A Formal Model for Components with Behaviours 15
1. Port-Based Components and Behaviours 15
2. I/O-Transition Systems (IOTS) with Queues 22
3. Components with (A)Synchronous Communication 26
4. Implementation using Message-Oriented Middleware 31

Chapter 3. Behavioural Neutrality in Synchronous Assemblies 35
1. Syntactical Reduction of Synchronous Assemblies 35
2. Port-Based Views of Component Behaviour 39
3. Application to the Compressing Proxy System 43
4. Discussion and Related Work 44

Chapter 4. A Theory for Refinement and Compatibility 47
1. Input-Persistent I/O-Transition Systems (PIO) 48
2. Asynchronous Communication 58
3. Compatibility and N-ary Composition 64
4. Greybox Refinement and General Properties 66
5. Discussion and Related Work 73

Chapter 5. On the Verification of PIO Systems 75
1. Verification Based on Finite-State PIOs 76
2. Correspondence with Communicating Finite State Machines (CFSM) 81

Chapter 6. Frames for the Specification of Component Behaviours 99
1. Frame Specifications for Port-Based Components 99
2. Compressing Proxy Revisited 103
3. Supporting Component-Based Development 107
4. Port-Based Frame Verification of Communication-Safety 110
5. Related Work 112

Chapter 7. UML2 – Applied Features and Extensions 113
1. Static Structure of Components and Assemblies 113
2. Component Behaviour and UML State Machines 117
3. Frame Specifications and UML Protocol State Machines 119
4. From State Machines to Transition Systems 120

Chapter 8. The Common Component Modelling Example (CoCoME) 125
1. Modelling of the CoCoME 125
2. Static Structures 126
3. Component Behaviours and their Translation 129

4. Hierarchical Component Behaviours 131
5. Frame Specifications of Simple and Composite Components 136
6. Analysis and Proof Obligations 137

Chapter 9. Conclusion 143
1. Related Work 143
2. Evaluation and Prospects 152

Bibliography 155

CHAPTER 1

Introduction

1.	**Component-Based Software Engineering**	**1**
2.	**Component Models and Distributed Systems**	**4**
3.	**Contribution and Overview**	**9**

The thesis starts with an introduction into component-based software engineering as a sub-discipline with distinct characteristics which allows us to separate it from other paradigms such as, for instance, object-oriented software engineering. In particular, we identify the specifics of component-based development to which the provision of an appropriate software component model is fundamental. A software component model defines what exactly components are and how they have to be composed in order to support the construction of larger systems. Software component models differ in variable classes of systems. Within this thesis we examine concurrent message-passing systems with synchronous and asynchronous message exchange and develop a formal software component model with a focus on modelling and behavioural analysis. After a discussion of what we believe constitutes a formal software component model, we summarise the contributions of this thesis and provide an overview of the single chapters.

1. Component-Based Software Engineering

Component-based software engineering (CBSE) has emerged as a sub-discipline of software engineering aiming at the development of reusable components and component-based systems. Reusability might be achieved by composing systems out of prefabricated components which can be considered as key aspect of the paradigm underlying CBSE. In this context it is an important goal to be able to compose *without* reference to a certain implementation [**Som06**, Chap. 19]. Components are supposed to be delivered by third-parties, or at least by independent project teams, as binary units so that there is no necessity to compile a component before it is composed with other components to be integrated into a larger system. Therefore, it is important to develop components as encapsulated entities that are equipped with a precise specification of their static structure as well as their dynamic behaviour.

CBSE is rooted in a promise of software engineering with off-the-shelf components, developed by third-parties and composable for system construction without too much knowledge of component implementation details but with enough knowledge to judge appropriateness with regard to given requirements [**NR68**]. It turned out that it is extremely difficult to find the appropriate abstraction level for the unambiguous description of components. We believe that there is still no consensus of what constitutes a *complete* component description including an integration of control- and data-related specifications as well as an integration of non-functional properties. However, focused research on each of these issues is promising and it seems that an integrated description is not out-of-reach. In contrast, agreement seems to exist concerning the description of functional component properties that relate to the control behaviour of components. Control behaviour of components is usually specified by protocols describing temporal orderings of message exchange

or method calls. Then the relation to an implementation is either by a formal implementation relation or by code generation. Since implementations are equipped with data, these relations usually include an abstraction step.

Regardless of the concrete relation between specification and implementation, the specification of control behaviour is used for two complementary development concerns in CBSE. On the one hand, specifications are composed to compute the behaviour of an assembly of components. Moreover, they are abstracted, hiding internal details, to effectively construct behavioural specifications for hierarchical component-based systems. On the other hand, these descriptions are also used as a design specification for the *independent* development of components as mentioned above. Therefore, it is crucial to obtain a precise understanding of the interplay between composition and independent refinement respectively implementation. Formal software component models examined hereafter (cf. Sect. 3) aim at providing exactly such an understanding.

Besides system construction, also maintenance and evolution focus on components and their specification. Components are either replaced by new versions or existing components are modified. In both cases it is important to know about the effects of the modification before any part of the original system is indeed touched. Unambiguous specifications again may serve as a helpful tool. Altogether, CBSE is characterised as follows:

(1) Independent development of reusable components
(2) System development by assembly and hierarchical composition
(3) Maintenance and evolution by substitution (and adaption) of components

Component development should be independent to support reuse, be it internal reuse within some given project or external in the sense that commercial prefabricated components are used. Moreover, independent development also leads to looser coupling of an integrated system and therefore helps in understanding the structure and behaviour of large component-based systems. The key to CBSE is composition. Systems are developed by composition of components and connectors, resulting in what is called an assembly. Assemblies in turn can be encapsulated within a composite component that can again be used for composition with other components and connectors. In this way, CBSE features a form of hierarchical composition. Finally, maintenance and evolution of a component-based system proceeds mainly by component substitution. Of course, also adaption of existing components, or more precisely, of existing component implementations plays an important role. However, such an adaption might also be understood as a substitution using a new component with an equivalent specification, but modified implementation.

Component-Based Development. We refer to the development process embedded within CBSE by *component-based development (CBD)*.[1] In general, system construction often follows either a top-down approach where initial system requirements are decomposed into parts that are refined stepwise until an executable system is finally obtained, or bottom up where first the parts of a system are designed in detail and then the parts altogether are composed to achieve an implementation with respect to given requirements. CBD often follows a mixture of top-down and bottom-up steps during system development. While one starts with general system requirements and proceeds top-down to construct a component architecture appropriate for the given requirements, one also searches for existing components that might be reusable for the construction of the given system. If such a component indeed exists, it is integrated into the existing architecture in a bottom-up way. The following (simplified) steps might be considered to be characteristic for component-based system development; cf. [**Crn03**, p.155] or [**Som06**, Sect. 19.2]:

[1] CBD is sometimes also denoted as component-based system development (CBSD). The notion of CBSE is usually used in the broader sense of an engineering discipline; cf. [**Crn03**] for instance.

(1) Identify usable components (e.g. repository)
(2) Select components; refine or modify requirements
(3) Adapt or implement missing components
(4) Compose and deploy components (after validation)
(5) Maintain system by replacing or modifying components

The identification of usable components is guided by the initial requirements for the system under construction. Since CBSE aims at reuse of existing components, this activity appears quite early in the development process. The components are selected, and according to component availability, requirements might be modified (if a system is constructed faster due to reused components, clients might be willing to accept modifications of their initial requirements). It is unlikely that all required components are already available, hence it is necessary to adapt existing or implement new components in order to fill gaps due to missing components. After that, the final set of components is validated, composed, and deployed. Note that the previous step of component implementation may already involve component composition for the realisation of composite components. Finally, and as mentioned in the more general context of CBSE above, maintenance is characterised by substitution of existing components. Either we replace a complete component, or we may modify its implementation.

Fundamental to the CBD process is the general component model chosen as an appropriate means to model the given system under construction. More precisely and following Lau and Wang [**LW07**], it is the *software component model* which plays a key role during component-based development. A software component model defines the syntax, the semantics, and the composition of components relating to the following questions: How are components represented? What exactly are components? How are components composed? Software component models (SCM) are supposed to answer these questions for the complete life cycle of a component: design, deployment, and runtime.[2]

Concerning the design of components, there is a long-standing research direction in the context of architectural description languages (ADL). Research within this area usually focused on the design and analysis of systems composed of "architectural units". Two prominent ADLs, Darwin [**MDEK95**] and Wright [**AG97**], using components and connectors as their architectural units are considered more detailed in our comparison with related work in Chap. 9. Opposed to the design focus of ADLs, there exist industrial component-based frameworks with an implementation focus such as CORBA [**OMG**] or Microsoft .NET [**Mic, TGG$^+$05**]. Since the component models of these frameworks do usually not fully support the component concepts underlying the models of ADL approaches, a number of frameworks, e.g. Fractal [**BCL$^+$06**] or SOFA [**BHP06**], and architectural programming languages, e.g. Java/A [**Hac04**], ComponentJ [**SC00**] or ArchJava [**ACN02**], emerged that try to close the conceptual gap between design and implementation of component-based systems.

All of the component models mentioned above have in common that a particular software component model is defined and used. However, since the concepts of the models differ sometimes considerably, one may ask to what extend the main CBSE characteristics stated above are indeed supported within the respective approach. Lau and Wang discuss this issue with a focus on the life cycle of components and use a slightly extended list of characteristics that is already biased by their notion of *software* component model and therefore provides a useful reification of the more general characteristics for CBSE: "[..] components should be preexisting reusable software units", "[..] components should be produced and used by independent parties", "[..] components should be composable into composite components", "[..] it should be possible to copy and instantiate components"

[2]In terms of programs one may think of a program design, an installed program and a running program

and "Composition means [..] also a systematic approach to system construction" [**LW07**, p. 709].

The first three requirements directly translate the general characteristics of reusability, independent implementation, and hierarchical composition as discussed in CBSE context above. The remaining aspects, however, provide more detailed requirements. First, instantiation of components calls for a notion of declarations to be used for system construction. For this reason, component type descriptions are needed that may be (re)used within a declaration similar to attribute declarations within a class of some object-oriented language. Second, the aim of a "systematic approach to system construction" based on composition obviously entails their requirement on hierarchical composition with composite components, but may additionally be considered to be a call for the general support of top-down and bottom-up approaches to system development. Due to their explicit distinction between design-time, deployment and runtime entities, a systematic approach should provide a link between deployed and running entities (at the bottom of a system development) and their design specifications (at the top of a system development). Therefore, if an entity on either level is touched (e.g. replaced or modified), the consequences for the other level should be clear. Such a notion of software component models gives rise to characteristics of component-based development along a certain SCM as shown in Fig. 1.1.

(1) Independent development of reusable component types
(2) Reusability of component types along declarations
(3) System development by assembly and hierarchical composition
(4) Maintenance/Evolution by substitution (and adaption) of component types
(5) Support for top-down and bottom-up system development

FIGURE 1.1. Characteristics of component-based development

Lau and Wang point to the necessity to have a composition theory available that allows a formal reasoning about composition [**LW07**, p. 711]. One of the aims is to be able to predict the effect of component composition. We think that this remark directly leads to the notion of *formal* software component models as discussed in Sect. 3 below. A formal software component model is a SCM that supports formal reasoning about the behaviours of components and their composition.

2. Component Models and Distributed Systems

Any software component model must be precise about the mechanisms of component composition within the particular model. Depending on what a component exactly consists of, such a composition means that components are somehow connected allowing to interact respectively to communicate with each other. Note that here and in the following text we will use *interaction* and *communication* synonymously. There is no hidden means of interaction in component-based systems. Components must provide a public interface[3] that describes the possibilities to interact with that component. Due to this strict encapsulation, it does not seem to make much difference if we talk either of interacting or of communicating components.

One particular class of systems whose development might take the most advantage out of a component-based approach are distributed systems. Andrew S. Tanenbaum provides the following loose characterisation of this class of systems: "A distributed system is a collection of independent computers that appears to its users as a single coherent system" and gives a schematic view as depicted in Fig. 1.2 [**TM07**].

One of the important characteristics of distributed systems is the possibility of uniform interaction between different applications despite possibly different operating systems or particular network services. *Middleware* aims, amongst others, at providing this means

[3]Here we do not yet refer to the technical notion of UML or Java interfaces

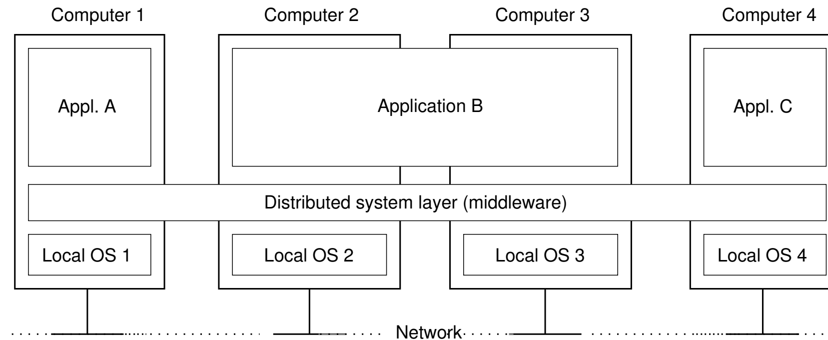


FIGURE 1.2. Distributed systems and middleware (from [**TM07**, Fig. 1-1])

and is realized by a software layer that spans over all machines which should be integrated into a distributed system. The software then provides communication facilities for the particular applications hiding away heterogeneous operating systems or network services.

The given figure shows that systems of this kind fit well with the characteristics of component-based systems. The different applications correspond to components and the middleware providing the communication facilities implements connectors that are used by the particular applications to communicate with each other. The distributed application B could be modelled by a composite component that consists of two subcomponents executing on computer 2 and 3 respectively.

Communication in distributed systems is always based on message passing [**TM07**, p. 115] which is mainly due to the fact that there is no shared memory in a distributed system. Applications send and receive messages using the given middleware. Even though the availability of concrete communication facilities depends on the implementation of the middleware, there are important general properties of communication that can be discussed independently from concrete implementations. Amongst others one may distinguish *synchronous* and *asynchronous* communication. With synchronous communication a "sender is blocked until its request is known to be accepted" whereas with asynchronous communication the "sender continues immediately after it has submitted its message for transmission" [**TM07**, p. 125].[4] Two important corresponding communication mechanisms are *remote procedure call* (RPC) and *message-oriented middleware* (MOM). The former aims at providing a more convenient way of interprocess communication than the latter. In fact, message-oriented communication is considered as a low-level mechanism that might be used as a mechanism to realize remote procedure calls. The basic idea of RPC is simple: instead of calling a procedure on the same machine, it is possible to call a procedure on a different machine. There are a number of technical difficulties, but the basic concept remains. Concerning synchronisation, as in local calls, the sender is blocked until the result returns. There is also a notion of asynchronous RPC that we do not further elaborate here and refer to [**TM07**] instead. The notion of *remote method invocation* (RMI) is principally equivalent to RPC. The main difference is that RMI applies to object-based distributed systems. Thus, methods are invoked on remote objects using global object references. While RPC is usually used to enable synchronous communication between distributed applications, message-oriented middleware allows for asynchronous communication that completely decouples the sender and receiver of messages. The receiver of a message must not even exist at the time some application sends out a message. In case

[4]Note that this notion of synchronicity could be interpreted in the sense of process algebraic rendezvous synchronisation: if several outputs are available, the sender is blocked before committing to any output and released as soon as it is known which outputs are accepted

of so-called persistent communications, the middleware will store the particular message until some receiver comes back to live again.

Message-oriented systems usually allow for message exchange following a point-to-point or a publish/subscribe pattern. For instance, Java Message Service (JMS) [**Mic02**], a specification for message-oriented middleware implementations in Java, exactly considers these two models of message exchange. With point-to-point communication the sender and receiver of messages use queues for buffered communication. With the publish/subscribe model so-called topics are used to which a receiver subscribes. In case a sender publishes a message related to this topic the receiver is notified accordingly. A more detailed description of these modes of communication is given in [**Mic02**], and more concretely in the context of the reference implementation of JMS in [**Mic07**].

The publish/subscribe model is directly related to the support of *group communication* in distributed systems. Following the terminology used by [**Kaa92**], group communication denotes sending of messages from 1 to n destinations; with *broadcast*, a group communication is implemented that requires all destinations to receive the message, while *multicast* requires reception only by those that indeed had announced interest before.

As apparent from a further study of the goals of distributed systems [**TM07**, pp. 3–15], the communication mechanisms mentioned find their counterpart in the behavioural descriptions of components and their composition. Formal software component models provide a perfect match with the goal of "openness". An *open distributed system* is a system that provides services through interface definitions. These specifications must not only cover syntactical issues, such as available operation signatures, but they are also expected to provide semantical information such as the effect of a particular service. In [**TM07**, p. 8] the importance of proper interface specifications in order to support "interoperability" and "portability" is stressed. The former relates to interaction between different applications that relies solely on the interface specifications, and the latter refers to the possibility to reuse an application in distributed systems that implement the same interfaces. We would claim that these characteristics are covered by independently developed component types that are reused and composed based on their behavioural specifications only. Portability may additionally require support for bottom-up system development which allows to first develop a single component in great detail and then to put this component into a larger context again by composition that relies on the specification of the particular component.

Since, in comparison with RPC, message-oriented communication is considered the more basic, low-level communication mechanism [**TM07, Kaa92**], labelled transition systems seem to provide an appropriate formalism for a precise abstract representation of these specifications. Indeed, transition systems are an established and widely used operational model for the modelling and analysis of parallel systems, in particular for message-passing systems. Baier and Katoen define two basic mechanisms for the description of communication in the context of transition systems: messages are either transferred synchronously using a handshake mechanism (rendezvous) or asynchronously by buffered message exchange using channels [**BKL08**, pp. 35 ff.]. Analysis of systems with buffered message exchange is notoriously difficult. If the channels are assumed unbounded (for more convenient modelling purposes), the underlying transition systems are potentially infinite-state making nearly any verification problem undecidable [**BZ83**]. However, even if the channels are bounded by some fixed capacity, the state-space soon becomes too large to apply finite-state verification techniques effectively. This is one of the reasons why synchronous communication based on a handshake mechanism, as prevalent in most process algebraic theories, is an important means of modelling and analysing the behavioural aspects of component-based systems. The transition systems resulting from the synchronous composition of communicating components consist only of successful communication traces and thus are usually much smaller with regard to the global state space. Note that "successful"

is defined independently from the communication direction. If a sender may choose non-deterministically among a set of alternative output transitions, all of these outputs are considered for synchronisation with corresponding input transitions of a potential receiver. In this vein only the successful synchronisations show up in the composed transition system. Since these transitions correspond to the potential communications between components, the synchronous mechanism yields a correct but incomplete representation. In particular, when it comes to analysing communication errors like messages sent that are not received, this issue of incompleteness is of major importance.

The combination of the characteristics of concurrent message-passing systems such as distributed systems and a formalisation of communication based on transition systems yields a *formal component model* with a precise definition of static entities and their means of interaction. Components represent possibly hierarchically structured applications that communicate with each other via synchronous and asynchronous connectors. Communication consists of sending and receiving messages, and is described by behavioural specifications based on transition systems. The composition of components and connectors is called an assembly whose behaviour is a derived transition system. Composite components use assemblies for their implementation again showing a derived behaviour that is based on the behaviour of the encapsulated assembly.

The following example illustrates a specification of a simple component-based system, exemplifying the concepts introduced so far. In particular, we would like to illustrate the relation to distributed systems as an example for an implementation within the general class of concurrent message-passing systems.

Bank/ATM System. In the following we specify a simple Bank/ATM system using the UML2 for static structure specification and UML state machines for the behavioural descriptions. We do not aim at illustrating or explaining available modelling features in detail. We rather would like to give a small example to clarify some of the main concepts of component-based systems introduced so far.

Figure 1.3 shows the specification of the system. A composite component with type BankAtm contains an assembly of two simple components with types Bank and Atm used for the component declarations bank:Bank and atm:Atm. Ports specify typed interaction points of a component. The simple component types Bank and Atm have port declarations a:AtmCom, m:Mgt and s:Srv respectively that are connected with an asynchronous assembly connector called ab and type Bat and an anonymous delegate connector. The two simple component declarations and the assembly connector form an assembly with open port m:Mgt. The provided and required interfaces of the port types AtmCom and Srv with their operations are depicted using the UML ball-and-socket notation on the right-hand side of Fig. 1.3. The port type Mgt is specified analogously and not further described. If two ports are connected by an assembly connector, the provided interface of one port has to match the required interface of the other and vice versa. The open port of the assembly is connected to the port declaration r:Mgt of the composite component BankAtm. Here, we reused the same port type, but in general one may also use a different type as long as the particular interfaces match.

Our behavioural descriptions are based on transition systems with distinguished input, output, and internal actions focusing on the temporal order of messages sent and received interspersed with local internal computations. For the concrete specification we use a simple subclass of UML state machines. Hence, any existing UML modelling tool can be used for the design of port-based components. Behaviours for the component types Bank and Atm are given in Fig. 1.3. Input and output messages are indicated by $p.m/$ and $/p.m$, respectively, where p is the port name on which the message m is sent or received respectively. The behaviours are slightly simplified as they do not show internal actions and the bank behaviour does not make use of the port m:Mgt. The behaviour of the assembly and

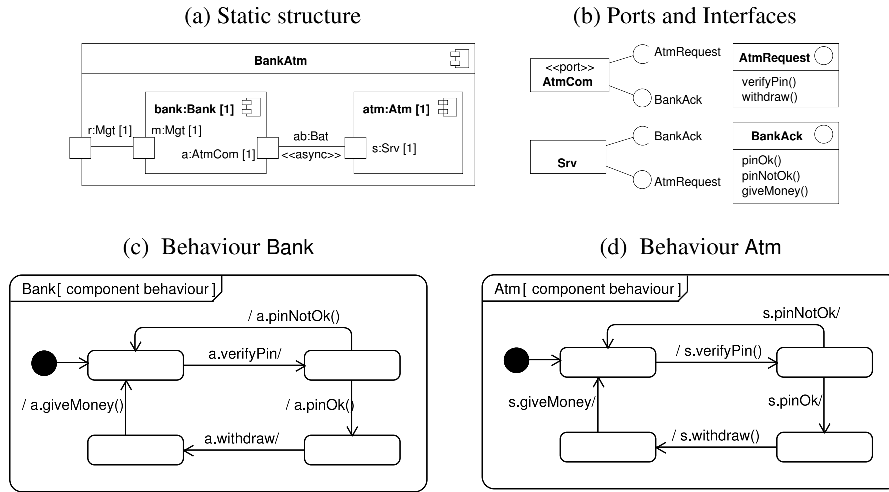


FIGURE 1.3. Specification of a simple Bank/ATM system

the composite component is derived from a combination of the behaviours for the components Bank and Atm. The behaviour at the ports m:Mgt and r:Mgt, however, is trivial due to the simplified presentation of the bank behaviour sending and receiving messages via port a:AtmCom only.

Figure 1.4 illustrates a possible realisation as a distributed system. Note that the figure is not supposed to specify a formal deployment view to the given system, but merely takes up the schematic view of Fig. 1.2 neglecting any questions related to types, instances, or concrete connections. The composite component BankAtm is realised as a distributed application. The application Report is assumed to be realised by a component that connects to the open port r:Mgt of BankAtm.

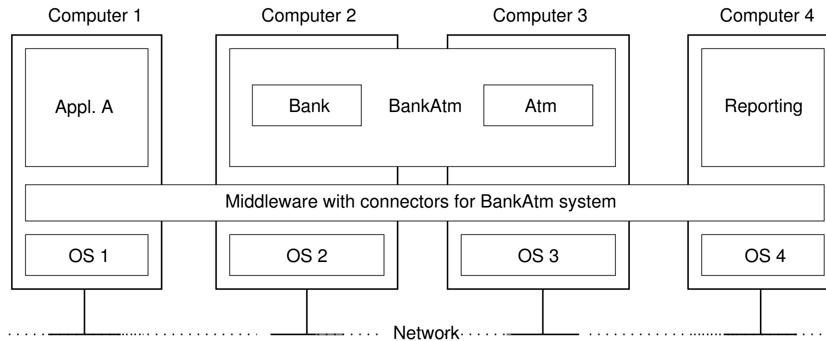


FIGURE 1.4. Bank/ATM distributed system

Now immediately the important question arises about what exactly is the role of the component behaviours given above with regard to such a realisation? Obviously the behaviours should be used for formal reasoning about the given system, but what are the properties of interest? What are notions of correct implementations allowing to replace a component of the given system without effect on the remaining components? These and similar questions will be addressed in the context of this thesis on the basis of an appropriately defined and formally precise software component model.

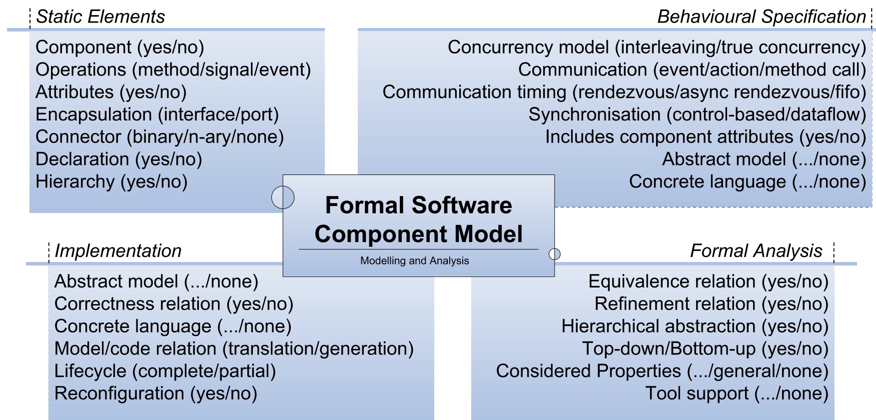


FIGURE 1.5. Aspects of formal software component models

3. Contribution and Overview

Within this thesis we develop a formal software component model with a focus on modelling and general behavioural analysis. Figure 1.5 provides an overview of what we think are fundamental categories and aspects in this context (values given in parentheses are used for a comparison with related work in Chap. 9). The given categories can be aligned to the notion of software component model as discussed above (cf. Sect. 1). The static elements define *how components are represented*, and together with behaviour and implementation it is also defined *what is represented*. Moreover, the execution model underlying the behavioural aspects additionally defines *how components are composed* and hence the categories indeed cover all information that constitute a software component model in the sense of Lau and Wang. However, we draw a sharper line between component and *software* component models, since we require an explicit distinction between specification and implementation of component behaviour to qualify as a software component model. In contrast, our requirements on support for formal analysis again may be put into line along the notion of *composition theory* [**CSSW04**] mentioned by Lau and Wang; cf. [**LW07**, p. 711]. A composition theory formally describes the result of applying component composition. We understand *composition* here in a broader sense, including a form of hierarchical composition using blackbox views of component behaviour.

As evident from Fig. 1.5, formal software component models integrate aspects from different but closely related domains. For instance, we use the UML2 [**RJB05**] for the concrete specification of structural and dynamical aspects of port-based components. Inspired by ROOM [**SGW94**] we favour port types as an explicit means for structural specification of interaction points, which is in contrast to strictly interface-based component models. Our component model is a formally defined software component model aimed at being used for architectural design specifications of concurrent systems that rely on message-oriented communication. This choice is directly reflected in our execution model that uses an interleaving model of concurrent message exchange. In contrast to most other approaches we do not restrict to synchronous rendezvous communication, but also consider asynchronous FIFO buffered communication. The behavioural model is backed by input-persistent I/O-transition systems (PIO) being a specific class of modal I/O automata [**LNW07**]. Implementations are represented abstractly by completely persistent PIOs coinciding with I/O-transition systems that in turn are based on interface automata of de Alfaro and Henzinger [**dAH01a**]. To illustrate specifications with a concrete language we will use a restricted class of simple UML state machines for implementations and protocol state machines for specifications. In order to illustrate implementations along a concrete

programming language we discuss the implementation of port-based components based on the Java Message Service specification JMS [**Mic02**].

We define equivalence relations similar to process algebraic approaches such as Wright, Darwin, and more recently PADL [**BCD02**]. Most importantly, we define refinement relations for specifications related to [**dAH01a, LNW07**] as well as a correctness relation between specifications and abstract models of implementations methodologically related to [**PV02**]. Finally, we aim at providing *formal* support for component-based design which in particular means to allow hierarchical abstraction, to support top-down and bottom-up approaches to system design, and to support "component-wise evolution" as discussed in [**dAH01b**] and similar also in [**PV02**]. As a concrete property being preserved during top-down or bottom-up development steps we consider communication-safety. This property implies the absence of architectural incompatibilities in the sense of [**BOR04**] or [**AP05**]. Concerning general properties we discuss the preservation of safety properties based on finite and infinite execution traces of a system, showing similarities with Tracta [**Gia99**].

Even though each of the mentioned aspects is covered by a number of related works, a formal software component model addressing an integrated view on the different aspects together with support for asynchronous message exchange as prevalent, for instance, in distributed systems is still missing. Especially, the formal relation between abstract models of behaviour specifications and models of implementations is often neglected. However, a precise theory for specification refinement and implementation correctness is crucial for a proper support of hierarchical system construction in the context of component-based system development. Moreover, a statement on specific and general properties that are preserved by correct hierarchical system construction is important to indicate the applicability of the developed theory. Here, the integration of asynchronous communication imposes particular difficulties. The contributions of this thesis in these aspects are as follows:

(1) An integrated formal software component model is developed that covers both, static and dynamic aspects of hierarchical system development. Ports play an important role for the structural definition of components' interaction points.
(2) Models for the specification and implementation of component behaviours are distinguished and a theory for refinement and implementation correctness is developed. The models take asynchronous communication into account by an integration of FIFO queues for buffered message exchange. It is shown that hierarchical component-based system development is properly supported by the formal part of our component model.
(3) Notions of synchronous and asynchronous compatibility are developed, exemplifying specific properties that are of interest in the context of formal behavioural analysis. The preservation of safety properties is examined as an example for more general properties. The verification of asynchronous compatibility in closed systems is addressed by a criterion based on synchronous compatibility. General verification of systems with asynchronous communication is addressed by a translation to Communication Finite State Machines (CFSM).

Overview. Table 1.1 shows an overview of the definitions for syntax and formal models used by our component model. The numbers in parentheses refer to the corresponding chapter. An abstract syntax and formal model for static specifications is defined by a meta-model and a set-theoretical (algebraic) model in Chap. 2. For dynamic system aspects we distinguish the implementation and specification of component behaviours by "behaviours" and "frames". The formal model of behaviours uses I/O-transition systems (IOTSs) and is developed in Chap. 2. Before our component model is extended with an abstract syntax for frames in Chap. 6, we develop a refinement theory based on input-persistent I/O-transition systems (PIOs) in Chap. 4. The concrete syntax is defined in Chap. 7, using UML components with ports for static structures, and simplified variants of UML state machines (STMs) and protocol state machines (PSMs) for behaviours and frames. In the following

we give a more detailed overview of the mentioned and the remaining chapters of this thesis.

Component	Abstract Syntax	Formal Model	Concrete Syntax
Static	Metamodel (2)	Algebraic model (2)	UML Components (7)
Dynamic	Behaviour (2) Frame (6)	IOTSs (2) PIOs (4)	UML STMs (7) UML PSMs (7)

TABLE 1.1. Syntax and formal models

In *Chapter 2* we develop a formal model for port-based component behaviours with synchronous and asynchronous communication. The general concepts are defined by a metamodel and illustrated using the example of a compressing proxy system. The metamodel is complemented by an algebraic model, making precise the core structural elements and behavioural features of port-based component systems. Component behaviours are formally represented by I/O-transition systems which are equipped with basic operators and equivalence relations as known from process algebraic languages such as CCS [**Mil89**] or FSP [**KM06**]. Message queues are encoded by I/O-transition systems and composed with component behaviour, resulting in systems with FIFO buffered message exchange. On this basis we define a formal model for components with behaviours that communicate by synchronous and asynchronous message exchange. We include guidelines for implementations based on message-oriented middleware using JMS and its reference implementation Open Message Queue [**Mic07**].

Support for the construction and behavioural analysis of large scale component-based systems that communicate solely by synchronous communication is developed in *Chapter 3*. In this way, also an application of the formal model of Chap. 2 is illustrated. The crucial concept is a notion of neutral component behaviour. Component behaviour can be considered to be *neutral* within a given assembly if its composition does not show an effect on the assembly behaviour apart from the synchronisation of shared transitions without loosing any transition of the original assembly behaviour. A major result of this chapter is an algorithm that shows how to remove neutral components for the computation of a syntactically reduced but behaviourally equivalent assembly. A reduced assembly may then be used to implement a given composite component in a more efficient way. Moreover, we show that it is sufficient to perform neutrality checks using the projection of a component behaviour to one of its ports, provided that the projected behaviour is a weakly deterministic I/O-transition system. The port projection is usually a simpler and smaller transition system, at least after minimisation with respect to greybox respectively blackbox equivalence. The approach is illustrated with the compressing proxy system introduced in Chap. 2.

Chapter 4 elaborates a theory for refinement and compatibility of PIOs with synchronous and asynchronous communication. In order to support hierarchical system development and analysis of component behaviours, the refinement relation should satisfy fundamental properties such as reflexivity, transitivity, and compositionality. Moreover, the relation should, on the one hand, preserve as much properties as possible, but on the other hand leave enough freedom for concrete design decisions. Our study is motivated by the aim to detect behavioural incompatibilities as discussed in Chap. 2, taking into account the asymmetry of sending (output) and receiving (input) messages. We develop different notions of output compatibility with varying degrees of freedom for the interleaving of local, non-shared component actions. It turns out that, in order to preserve output compatibility, we need additional information that allows to couple the message transfer with the refinement of the transition that specified the transfer. On this account our approach extends I/O-transition systems to input-persistent I/O-transition systems (PIO) and defines a blackbox refinement relation that demands the preservation of persistent transitions. The

relation is denoted “blackbox” because it ignores differences of internal labels. PIOs can be considered as a special case of modal I/O automata (MIO) and then, blackbox refinement corresponds to weak modal refinement.

Our theory is required to cope with FIFO buffered message exchange. Based on an encoding of FIFO buffers we show that blackbox refinement is indeed transferable to asynchronously communicating components. Moreover, we develop a notion of asynchronous output compatibility that is a natural extension of synchronous output compatibility to an asynchronous setting and show its preservation by blackbox refinement. The resulting theory is an interface theory in the sense of [**BMSH10**], which is briefly discussed in Sect. 5 of Chap. 4.

Additionally, we define communication-safety as a notion of output compatibility for n-ary compositions of PIOs. We show that blackbox refinement of single transition systems preserves communication-safety of their composition and, moreover, investigate to what extend pairwise analysis of output compatibility implies communication-safety. Finally, we consider more general properties and a greybox variant of our refinement relation. Greybox refinement distinguishes internal labels. We show that greybox refinement preserves safety properties with regard to the communication traces of a composed system and briefly discuss counter-examples for the preservation of liveness properties.

In *Chapter 5* we consider foremost the verification of asynchronous communication-safety. Due to the modelling with unbounded FIFO queues, we must take into account that systems of buffered PIOs are potentially infinite-state systems which makes most verification problems undecidable. We investigate a criterion for asynchronous communication-safety in closed systems based on synchronous compatibility. An important additional assumption, input-separation of the underlying transition systems, is conceptionally related to single-threaded components. For the general verification support of buffered PIO systems we discuss a translation to Communicating Finite State Machines (CFSM) [**vB78**]. CFSM systems consist of finite state machines communicating with each other via unbounded, full-duplex, and error-free FIFO channels. Thus, there is a close relationship of systems of buffered PIOs and CFSM systems. We describe a translation of IOTSs to CFSMs and develop a CFSM correspondence of asynchronous communication-safety. After that we sketch the application of a symbolic CFSM-based approach to the verification of asynchronous communication-safety. Finally, necessary extensions to verify PIOs instead of IOTSs are briefly discussed.

With the basic formal model in Chap. 2 components are equipped with behaviours (formalised by I/O-transition systems) that are supposed to represent the implementation of a component. *Chapter 6* extends this model by a specification layer with so-called *component frames* using the theory developed in Chap. 4. Frames are formally related to behaviours by a correctness relation: the semantics of a frame is the class of all behaviours which are a blackbox refinement of the frame. We illustrate the completed component model using the compressing proxy system from Chap. 2. After that, we show that the model supports important characteristics of component-based development. We examine support for top-down and bottom-up design approaches and, moreover, show that component-wise evolution [**dAH01b**] is supported in hierarchical systems. Top-down design starts with a frame F that is refined to a composition of several frames. Then, support for top-down design allows to conclude that the composition of component-wise correct blackbox refinements yields a correct implementation of the original frame F. By this kind of support it is shown that the definitions of refinement and implementation relations work smoothly together.

Bottom-up design first details the implementation of single components. If the behaviours are correct, i.e. they are a blackbox refinement of the frame of the respective component, then support for bottom-up design allows to conclude that the behaviour composition yields a greybox refinement of the corresponding frame composition. In this way it

is shown that the implementation relation preserves "greybox properties" of compositions and thus supports, for instance, analysis of communication traces in composed systems.

Finally, we discuss a port-based approach for the verification of compatibility and communication-safety based on frames and port protocols illustrating the fundamental difficulty of making effective use of port protocols that hide too much of the component behaviour.

In *Chapter 7* we define a concrete syntax for our component model using UML2 components with ports, state machines and protocol state machines. We summarise applied UML2 features and explain the meaning of additional notations and stereotypes. We relate static structure elements with our formal model and discuss an informal pattern-based translation from state machines to transition systems. Based on this translation we define the relation between UML state machines and protocol state machines, and component behaviours and frames as considered in Chap. 6. In this way, we may apply our theory as a formal underpinning for the definition of a UML-based theory of refinement (protocol conformance) and compatibility.

Availability of a concrete specification language was of particular importance for the modelling of the Common Component Modelling Example[5] (CoCoME) in [**KJH$^+$08a**] . CoCoME was initiated as a modelling contest aiming at a comparison of existing component models based on a common requirement specification and reference implementation of a point-of-sale system. *Chapter 8* revises our initial submission and illustrates features of our component model that were not available at the time of writing [**KJH$^+$08a**]. The formal model sketched in [**KJH$^+$08a**] considered only synchronous communication along a rendezvous mechanism. It did not distinguish between behavioural specification (frames) and implementation (behaviours) of components and it did not feature a theoretical model for refinement and compatibility. We provide a brief introduction to the CoCoME requirements and elaborate on our static structure specifications using UML class and component diagrams, and on our modelling of component behaviours using UML state machines. We illustrate the modelling of more complex component behaviours using hierarchical UML submachine states and translate selected behaviours to IOTSs based on Chap. 7. We specify component frames for simple and composite components using UML protocol state machines and discuss the analysis of compatibility, implementation correctness and refinement on the level of the corresponding transition systems.

The main part of the conclusion in *Chapter 9* discusses related work with respect to the aspects of formal software component models given by Fig. 1.5. We consider as relevant for a more detailed comparison only those component models from the literature that have similar objectives like Sofa 2.0 [**BHP06**], GCM/Fractal [**HKR08, BABC$^+$09**], STSLib [**FR08**], FSP/Tracta [**KM06, GKC99**], and Wright [**AG97**]. We conclude with remarks on prospects, limitations and future work.

Publications. The thesis is based on publications as listed below. In (P1) we modelled and analysed the CoCoME using the Java/A component model. The study in (P2) focused on the detailed development of a behavioural theory using I/O-transition systems. Finally, in (P3) we extended the model to take into account asynchronous communication. In the following we briefly relate the publications to the chapters of this thesis.

(P1) Alexander Knapp, Stephan Janisch, Rolf Hennicker, Allan Clark, Stephen Gilmore, Florian Hacklinger, Hubert Baumeister, and Martin Wirsing. Modelling the CoCoME with the Java/A Component Model. In Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, (Eds.): The Common Component Modeling Example: Comparing Software Component Models, LNCS 5153, pp. 207–237. Springer, Heidelberg (2008).

[5] `http://www.cocome.org`

(P2) Rolf Hennicker, Stephan Janisch, and Alexander Knapp. On the Observable Behaviour of Composite Components. Electronic Notes in Theoretical Computer Science (5th FACS'08), Vol 260 (1) Jan 2010, pp. 125–153.

(P3) Rolf Hennicker, Stephan Janisch and Alexander Knapp. Refinement of Components in Connection-Safe Assemblies with Synchronous and Asynchronous Communication. Christine Choppy and Oleg Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 154–180. Springer, Heidelberg (2010).

The Java/A component model described in (P1) provided the basis for a metamodel in (P2) for components with (observable) behaviours and synchronous communication. Parts of the modelling of the CoCoME in (P1) appear in Chap. 8. The formal approach to deadlock analysis of (P1) evolved into the formal behavioural model based on I/O-transition systems in (P2), yet without asynchronous communication. The analysis techniques developed in (P2) appear in revised and aligned form in Chap. 3

(P3) extends the behavioural component model of (P2) with asynchronous communication. The resulting metamodel and algebraic model appears in revised and slightly extended form in Chap. 2. Furthermore we developed a compositional theory for refinement, defined a notion of asynchronous compatibility (called buffered compatibility) and showed for closed systems with two components the preservation of compatibility under further assumptions like input-separated component behaviours. However, the approach did not carry over to open systems. Even though we believe that a generalisation to arbitrary closed systems is possible, we did not pursue the approach of (P3). Instead, we developed a new theory using input-persistent I/O-automata that appears in Chap. 4. The criterion for asynchronous compatibility in Chap. 5 was inspired by a corresponding theorem in (P3).

CHAPTER 2

A Formal Model for Components with Behaviours

1. Port-Based Components and Behaviours	**15**
1.1. Concepts of Port-Based Components	16
1.2. Example: Compressing Proxy System	17
1.3. Composition and Behavioural Incompatibilities	20
2. I/O-Transition Systems (IOTS) with Queues	**22**
2.1. Transition Systems with Partitioned I/O-Labellings	22
2.2. Operators on I/O-Transition Systems	23
2.3. Greybox and Blackbox Equivalence	24
2.4. Encoding of FIFO Queues	26
3. Components with (A)Synchronous Communication	**26**
3.1. Ports, Components, and Connectors	26
3.2. Assemblies and Composite Components	28
4. Implementation using Message-Oriented Middleware	**31**
4.1. Java Message Service and Open MQ	31
4.2. Implementation of Port-Based Components	32

This chapter develops a formal model for systems of port-based components with synchronous and asynchronous communication. The general concepts are defined by a metamodel and illustrated using the example of a compressing proxy system. The metamodel is complemented by set-theoretical definitions stating precisely the core structural elements and behavioural features of port-based component systems. Component behaviours are formally represented by I/O-transition systems that are equipped with basic operators and notions of equivalence as known from process algebraic languages such as CCS or FSP. Message queues are encoded by I/O-transition systems and composed with component behaviour, resulting in systems with FIFO buffered message exchange. On this basis we define a formal model for components with behaviours that communicate by synchronous and asynchronous message exchange. We close the chapter with guidelines for implementations based on message-oriented middleware using the Java Message Service specification and its reference implementation Open Message Queue. Related formal component models are discussed in Chap. 9.

1. Port-Based Components and Behaviours

We consider components to be strongly encapsulated behaviours. Encapsulation is achieved by ports that regulate any interaction of components with their environment. Components can be hierarchically structured containing an assembly of components and connectors that link components using their ports. Interaction consists of synchronous or asynchronous message exchange and we therefore distinguish synchronous and asynchronous connectors. By synchronous communication we understand a rendezvous mechanism where sender and receiver synchronise on message exchange. In contrast, asynchronous communication works with FIFO-buffering where the messages issued by a sender are buffered and can be taken (and processed) later on by the receiver. Sending and receiving

messages with synchronous and asynchronous communication timing are basic primitives of general message-passing systems [**Tuc04**, Sect. 96.3.4].

1.1. Concepts of Port-Based Components. Figure 2.1 shows the metamodel of our component model. A *port* describes a (partial) view on a component. The operations offered by a port are summarised in its *provided interface*; the operations needed in its *required interface*. An operation declares the signature to be used for message exchange along this port. Ports as well as components and connectors are in fact considered as types that can be used in local declarations of components and assemblies respectively. A *declaration* consists of a name and a type.

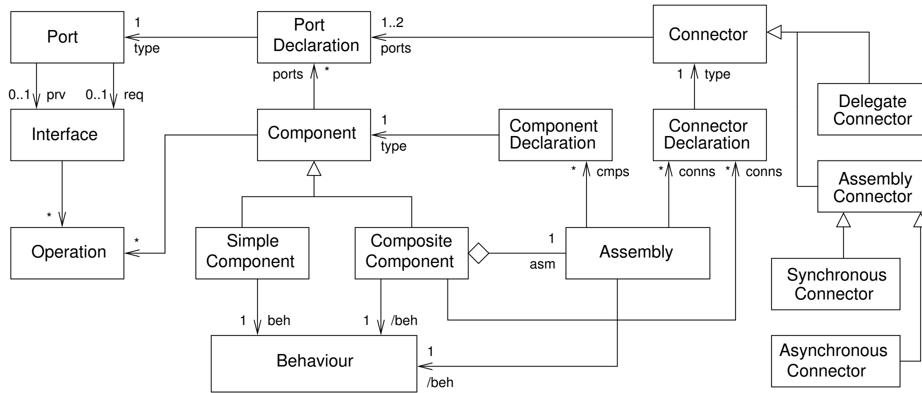


FIGURE 2.1. Component metamodel

There are two kinds of components, *simple components* and *composite components* which are abstracted in the metaclass *component*. Any kind of component has a set of *port declarations* that introduce locally unique port names with corresponding port types. Components can be used in component declarations for the construction of assemblies.

An *assembly* defines the internal structure of a composite component in terms of a set of local *component declarations* and local *connector declarations* using binary *assembly connectors* that link local components using their port declarations. Assembly connectors can be synchronous or asynchronous indicating which kind of communication timing is used for the message exchange along this connector. In a composite component, non-connected (open) ports of local components may be connected to so-called *relay ports* of the composite component, using declarations of *delegate connectors*. Unary connectors can be used to close ports of a component that should not be available, neither for connection nor for delegation. This feature is used in Chap. 3 to reduce assemblies with so-called neutral leaf components.

For each simple component a given *behaviour* is supposed to represent the implementation of a component. Behaviours describe the temporal order of receiving messages (input), sending messages (output) and executing local actions (internal). Composite components encapsulate an assembly as an implementation of their behaviour. Therefore, the behaviour of a composite component is a derived behaviour (indicated by the preceeding slash symbol in Fig. 2.1). An assembly in turn has an associated derived behaviour that depends on the behaviours of its local components and the types of its assembly connectors.

We use UML2 notation for the concrete specification of the static structure of component-based systems, in particular we make use of UML structured classifiers, components, ports, interfaces and connectors. Since all these entities except interfaces own so-called structural properties, for instance the port declarations of a component is a structural property, it is also allowed to specify multiplicities indicating how many instances of a component or port may exist at runtime [**RJB05**, property]. However, we will use singletons only

(multiplicity 1) and leave an extension of our formal component model supporting arbitrary multiplicities, especially an extension of the semantics on the level of global behaviours in hierarchically structured component-based systems, to future work. Moreover, we restrict the components of our models to binary communication. UML2 would allow for arbitrary n-ary connectors. However, these kinds of connectors would complicate our formal treatment of behaviours based on I/O-transition systems while being not strictly necessary from the semantical point of view: An n-ary connector is perfectly described by a component whose behaviour reflects the desired connector behaviour. Such a "connector-component" is then linked in between the n components along n binary connectors. Finally, note that we usually do not consider messages with different parameter values for the specification of a component behaviour while operations in interfaces in general may show formal parameters.

The behaviours will be described directly on a semantic level using (finite) I/O-transition systems (IOTSs, see Sect. 2) and represented by graphs with labelled transitions and states. We do not intend to propose a particular, e.g., process algebraic or state machine, syntax for specifying behaviours. In fact, any concrete syntax could be used as long as an interpretation using IOTSs is provided. This interpretation may also provide the construction of messages with different values according to the parameter types of the respective operation. In the graphical representation we indicate that a message m is received (input label, provided) by the visual representation $m/$ and, symmetrically, that a message m is sent (output label, required) by $/m$. Internal, local actions are represented by transitions labelled without slash. An initial state is indicated by an incoming arrow without source state.

1.2. Example: Compressing Proxy System. We illustrate our component model by a compressing proxy system.[1] An HTTP proxy server mediates connections between a web server and its clients. In order to increase network bandwidth, the proxy server may apply different compression techniques depending on the kind of information transferred. The proxy server distinguishes between textual (txt) and graphical (gif) data and applies different compression tools before sending the data further downstream.

We describe the static structure of the compressing proxy system by a composite component type CompressingProxy that consists of an assembly of three simple components with types Adaptor, GZip, and GifToJpg introduced by the respective component declarations adapt:Adaptor, gzip:GZip, and gifToJpg:GifToJpgas shown in Fig. 2.2. The adaptor receives uncompressed data upstream, delegates the textual data for compression to gzip, which employs the gzip utility, and the graphical data to gifToJpg, a tool to convert GIF- to JPG-images. The compression result will then be sent downstream.
The simple components as well as the composite component show port declarations such as t:TxtCompr or l:UpStream. The port declarations of composite components are called relay ports. Port declarations inside the assembly are connected by synchronous and asynchronous assembly connectors while delegate connectors link port declarations of inner components and relay ports of the composite component. Communication timing is specified by stereotypes «sync» and «async». The connector tz between t:TxtCompr and z:Zip is an asynchronous assembly connector, the connector gj is synchronous, and the anonymous connector between u:UpStream and l:UpStream is a delegate connector.

The provided and required interfaces of the port types of the compressing proxy system are depicted with the UML ball-and-socket notation on the right-hand side of Fig. 2.2.

[1]This example is to some extend identical with an example originally used by [**IU01**] (and also by [**BCD02**]) to illustrate an approach to prove deadlock-freedom in component assemblies.

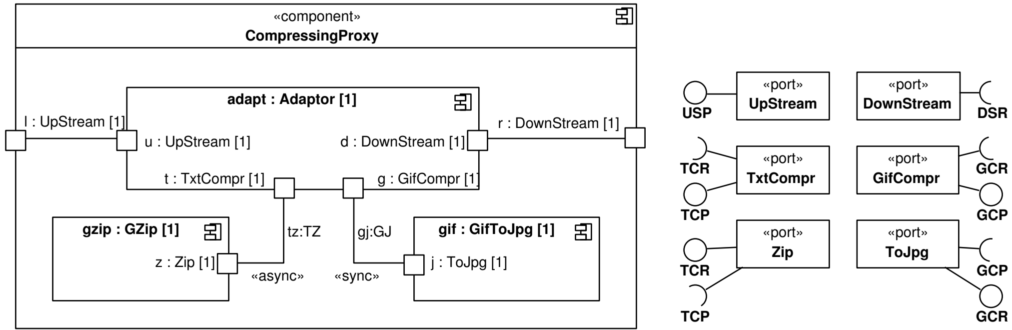


FIGURE 2.2. Static structure of a compressing proxy server

If two ports are connected by an assembly connector, it is required that the provided interface of one port is equal to the required interface of the other, and vice versa.[2] The operations of these interfaces are assumed to be given as follows:

$$\mathsf{USP} = \{\mathsf{openTxt}, \mathsf{openGif}\}, \qquad \mathsf{TCR} = \{\mathsf{txt}, \mathsf{endTxt}\}, \qquad \mathsf{GCR} = \{\mathsf{gif}\},$$
$$\mathsf{DSR} = \{\mathsf{comprData}\}, \qquad \mathsf{TCP} = \{\mathsf{stop}, \mathsf{zip}, \mathsf{endZip}\}, \qquad \mathsf{GCP} = \{\mathsf{jpg}\}.$$

We do not consider operations with parameters here, thus the operation names coincide with the messages used for communication.

Based on the static structure of the compressing proxy system, the informal description of its intended behaviour reads as follows: A proxy of type CompressingProxy receives stream-based data on its port l that is delegated to the port u of the contained component adapt. The adaptor distinguishes textual and graphical data received at port u and forwards data for textual compression via port t and graphical compression via port g. After receiving the compression result, the component sends the data further downstream using port d which is relayed to port r of the composite component.

The behaviour of simple components, which are basic building blocks of port-based component systems, are assumed to be given by IOTSs provided by the component implementor respectively designer. Figure 2.3 shows behaviours of the simple components Adaptor, GZip and GifToJpg. Since components communicate only via ports, any label representing a message received or sent has the form $p.m$ where p is a port name and m is a message according to the provided or required interface of the port. Behaviours also show internal actions represented by internal transition labels without prefix. Additionally we allow an anonymous internal action τ to reflect a possible internal choice of the component that can or should not be further detailed.

For instance, the behaviour of the component Adaptor shows label for receiving messages u.openTxt, u.openGif, t.zip, t.stop, t.endZip, and g.jpg, and for sending messages d.comprData, t.txt, t.endTxt, and g.gif. The only internal action is timeout. The adaptor component receives requests for either text or gif compression u.openTxt or u.openGif resp. at the upstream port u. After having sent an initial part of textual data t.txt via port t, there is a choice between sending further text until the end of the text is reached (signalled by t.endTxt), or the message t.stop is received, for example indicating a buffer overflow of the component attached for text compression. In the first case, the component waits to receive zip data (t.zip) until transmission end is signalled by t.endZip. In the second case,

[2] In general, one could use a more flexible condition so that the required interface of one port is included in the provided interface of the other one. However, it is technically more convenient to use the more restrictive condition from above.

[3] The adaptor behaviour corresponds exactly to the adaptor behaviour in [**BCD02**] if we remove the communication concerning the port g for compression of graphical data.

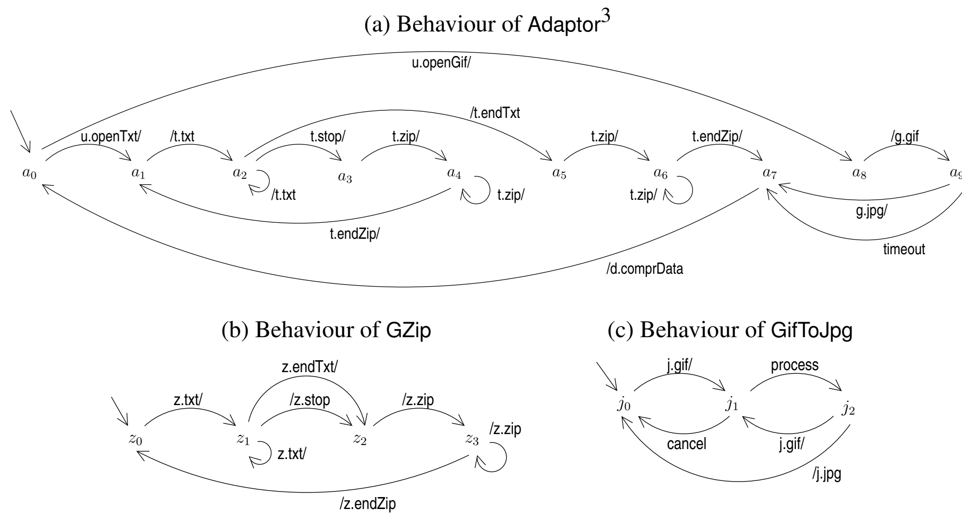


FIGURE 2.3. Behaviour of simple components

the component retrieves compressed data via t.zip and the communication proceeds until eventually the end of the text will be reached. In both cases a new communication can be started by sending again an initial piece of textual data to be compressed via t.txt. After the text compression is finished, the component sends the compression result comprData further downstream at port d.

Note that the communication between Adaptor and GZip is asynchronous, therefore the behaviours are not blocked until a message sent is received. Instead, the sender may continue immediately, analogously to the execution of local actions. An asynchronous connector is equipped with two FIFO queues, one for each communication direction. For the given protocol this means, e.g., that the adaptor may continue to send text even though the GZip component has sent a stop request in the meantime. This is not an immediate defect, since it is the FIFO buffer which is flooded by txt messages, not the communication partner. If we assume that the adaptor eventually receives the message stop then the communication may proceed indeed as required.

Graphical compression consists only of sending the GIF input further on via g.gif at port g and then to receive the JPG result back again. In case the compression takes too long, an internal timeout transition is triggered and the adaptor sends instead of a JPG the uncompressed data further downstream via d.comprData. The behaviour of the GZip component mirrors the corresponding part of the adaptor. In fact the design process following a bottom-up approach would probably start with the behaviour of the tool component GZip and then mirror this behaviour in the design of the adaptor component. The component GifToJpg shows an internal choice within its behaviour. After having received gif at port j, the component either processes the given GIF or returns without processing to its initial state. In the former case either the JPG result is sent back via j.jpg or another GIF is received at port g which transits back to the given internal choice. In this case a previous result is dismissed and JPGs are delivered only for the latest given GIF.

The behaviours of assemblies and composite components are derived behaviours. Assemblies use component declarations showing a name and a component type. In order to distinguish the different components inside an assembly, the labels of their behaviours are prefixed by the name of the component declaration, thus turning, e.g., u.openGif of the component Adaptor into adapt.u.openGif w.r.t. the declaration adapt:Adaptor. The overall

behaviour of an assembly shows both, the interaction behaviour of the composed components that are connected within the assembly via their ports, and the communication potential at the open ports of components that are not connected within the assembly. Hence, an assembly behaviour is a derived behaviour, given by the composition of the component behaviours of all declared components and connectors of the assembly.

The synchronisation within an assembly maps the labels of the ports connected by an assembly connector to the name of the connector, thus making them shared between the respective components. Transitions with shared labels synchronise and represent communication of the components via their connected ports. Open ports do not appear in assembly connectors, therefore those parts of component behaviours related to open ports remain unsynchronised. In general, we also consider the case where a port is neither open nor connected to another port. Technically, these ports are closed by applying unary connectors so that, altogether, the internal actions of an assembly behaviour comprise the synchronised transitions along the shared actions of its components and the unused actions of closed ports. The behaviour according to the remaining open ports is described by the unsynchronised input and output transitions of the component behaviours that are the only external actions of an assembly.

For instance the behaviour of the assembly underlying the CompressingProxy component is computed by the synchronisation of the component behaviours of adapt: Adaptor, gzip:Gzip, and gifToJpg:GifToJpg according to the connectors of their ports. The synchronisation label for the output label adapt.g.gif of adapt:Adaptor and the input label gifToJpg.j.gif of gifToJpg:GifToJpg is given in accordance with the connector gj by the label gj.gif. Now, both component behaviours may synchronise on transitions with the shared label gj.gif. If the assembly contains asynchronous connectors, the derived behaviour is a potentially infinite-state system, due to the unboundedness of the FIFO queues used to model buffered communication. For strictly synchronous assemblies, the composition can be computed and represented by finite-state techniques. Assuming that the assembly connector tz is synchronous, the mentioned assembly behaviour has 30 states and 65 transitions (including internal and τ-transitions). If we hide internal actions of components and use the minimisation procedure of observational equivalence, we obtain the simpler IOTS given by Fig. 2.4a. The minimised IOTS shows only synchronised and open transitions as well as those τ-transitions that could not be removed without invalidating observational equivalence.[4]

Finally, the behaviour of composite components is derived in such a way that the input and output actions of a composite component are delegated between its relay ports and the open ports of the assembly. The synchronised transitions of the assembly are considered to be internal labels of the composite component analogous to internal actions of simple components. When a component is put into the context of a new assembly, its internals are hidden, i.e. relabelled to the anonymous action τ. Figure 2.4b shows the resulting component behaviour of CompressingProxy, after hiding and minimisation with respect to blackbox equivalence under the assumption of tz being a synchronous connector.

1.3. Composition and Behavioural Incompatibilities. Fundamental to component-based design is a composition operator that allows to construct new components from an assembly of existing components (and connectors). Arbitrary composition might lead to a vast number of component interoperability errors concerning different aspects and properties of the underlying components. Becker et al. [**BOR04**] provide a classification of these errors in the context of general component characteristics comprising functional as well as non-functional aspects. In the centre of our interest are *architectural incompatibilities*

[4]Computed using the LTSA tool with an FSP encoding of the given behaviours.

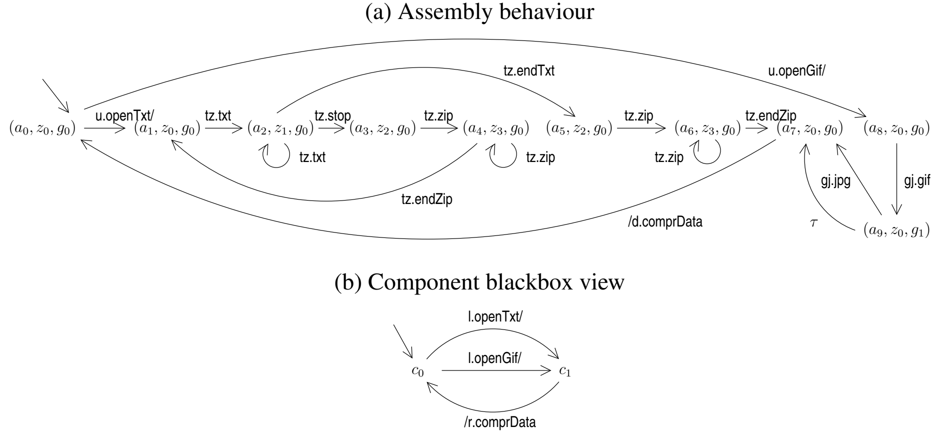


FIGURE 2.4. Assembly behaviour and blackbox view of CompressingProxy

which includes mismatches on the signature level of interfaces as well as incompatible assumptions on synchronisation of buffered or unbuffered message exchange. We denote the latter as *behavioural incompatibilities*.

While syntactical mismatches can be handled by static analysis of the interface specifications of a given component-based system, incompatibilities concerning the dynamic behaviours of components are usually much harder to cope with. For example detecting global deadlock states within an arbitrary assembly of components requires in general a complete search within the globally reachable state space of the composed behaviours. Therefore, it is of interest to examine compositional analysis techniques that allow to conclude global properties from modular component- or pairwise analysis. As an exemplary property we focus on a communication error that is justified by the component's local view on message exchange as follows. If a component receives a message, it is implied that there is some component in the system which had sent this message. Therefore, at the moment of message reception it seems that everything is correct within the running system. However, this does not apply to sending of messages. At the moment of message sending there is no guarantee at all that the message will eventually reach some receiving component within the given system.

Consider, for instance, the potential communication between the adaptor and the GIF converter component in Fig. 2.3. If the adaptor reaches state a_9, a timeout may occur and the behaviour proceeds to state a_7. Anyhow, the GIF converter might have reached state j_2 in the meantime and sends out the requested JPG with a transition to its initial state j_0. But then the message sent is not received by the adaptor, since state a_7 does not provide a corresponding reception transition. In order to detect this kind of architectural incompatibility we aim to develop appropriate notions of compatibility for synchronous and asynchronous communication within port-based component systems.

As a prerequisite we first need a more precise and formal model of component behaviours. Such a model should explain what behaviours are exactly and how they might be composed to construct larger systems. In particular, it should be able to cope with synchronous as well as asynchronous communication and it should provide appropriate notions of (observational) equivalence to allow the substitution of a given component by different but behaviourally equivalent components. The design of the model described in Sect. 2 takes these requirements into account. In Sect. 3 we formally relate this model to the concepts of Sect. 1.1. The formal component model also provides a detailed account on metamodel constraints mentioned in Sect. 1.1 only informally.

Furthermore, there should not be a must to redo analysis of architectural incompatibilities for any modification of a given component-based system. Instead, it should be possible to carry out analysis on the level of more abstract specifications of component behaviour and then conclude that a given property holds for any correct implementation. In this way, it is possible to substitute components as long as their implementation is correct with respect to the original specification. In Chap. 4 we develop a theory that supports this kind of reasoning. The theory is first developed independently from our concrete component model and then put into context in Chap. 6. This chapter also shows that our theory supports both, top-down and bottom-up approaches to component-based design. The former requires support for stepwise refinement of specifications and the latter requires an implementation relation that preserves absence of communication errors after substituting components by different but still correctly implemented components.

2. I/O-Transition Systems (IOTS) with Queues

Within this section we summarise definitions and basic properties for *I/O-transition systems* that are similar to the interface automata of de Alfaro and Henzinger [**dAH05**]. Minor syntactical differences are, first, the assumption on I/O-labellings to be *partitioned* inputs and outputs and, second, to allow besides internal transitions also an anonymous internal action τ for abstraction from internal actions. De Alfaro and Henzinger define composition and a refinement relation in order to investigate a particular notion of compatibility between interface automata, while we define composition, relabelling, hiding, and equivalence relations in order to formalise behaviours of components, assemblies and composite components. In particular composite components are not considered in [**dAH05**].

2.1. Transition Systems with Partitioned I/O-Labellings. The definition of I/O-transition systems with partitioned I/O-labellings relies on the (standard) notion of partitions. Let M be a set, $Z \subseteq \wp M$ is a *partition* of M if $\emptyset \notin Z$ and for $X, Y \in Z$, $X \neq Y$ implies $X \cap Y = \emptyset$ and $\bigcup\{X \mid X \in Z\} = M$. We write $\bigcup Z$ to refer to the set $\bigcup\{X \mid X \in Z\}$.

Definition 2.1 (IOL) *An* I/O-labelling *(IOL)* $L = (I, O, T)$ *consists of three mutually disjoint sets of* input *labels* I*,* output *labels* O*, and* internal *labels* T*; we write* $\bigcup L$ *for the set of labels* $I \cup O \cup T$ *and assume* $\tau \notin \bigcup L$*.*
A partitioned IOL $L = ((I_1, O_1), \ldots, (I_n, O_n), T)$*,* $n \geq 1$*, is an IOL where inputs and outputs are given by a partition* $\{(I_1, O_1), \ldots, (I_n, O_n)\}$*.*

Partitions of input and output labels are used for the formalisation of asynchronous communication in Chap. 4. Usually anything which holds for IOLs also holds for partitioned IOLs. If this is not the case it is explicitly stated, otherwise we silently assume that there is only a syntactical difference and do not mention partitioned IOLs explicitly.

Definition 2.2 (IOTS) *An* I/O-transition system *(IOTS)* $A = (L, S, s_0, \Delta)$ *is given by an IOL* L*, a set of states* S*, an initial state* $s_0 \in S$ *and a transition relation* $\Delta \subseteq S \times (\bigcup L \cup \{\tau\}) \times S$*. An IOTS* $A = (L, S, s_0, \Delta)$ *is* partitioned*, if* L *is partitioned.*

We use the following notations. If $A = (L, S, s_0, \Delta)$ is an IOTS, we write L_A, S_A, $s_{0,A}$ and Δ_A to refer to the labels L, the states S, the initial state s_0 and the transition relation Δ of A respectively. Moreover, if $L_A = (I, O, T)$, we write I_A, O_A and T_A to refer to I, O and T respectively. The transition labels of an IOTS A are given by $\mathcal{L}(A) = I_A \cup O_A \cup T_A \cup \{\tau\}$; the *silent (internal) labels* are given by $\mathcal{T}(A) = T_A \cup \{\tau\}$.

Definition 2.3 (Reachable states) *The* reachable states $\mathscr{R}(A)$ *of an IOTS* $A = (L, S, s_0, \Delta)$ *are inductively defined as follows:* $s_0 \in \mathscr{R}(A)$*; and if* $s \in \mathscr{R}(A)$ *and there is an* $l \in \bigcup L \cup \{\tau\}$ *and an* $s' \in S$ *with* $(s, l, s') \in \Delta$*, then* $s' \in \mathscr{R}(A)$*. For an* $s \in \mathscr{R}(A)$*, we write* $\mathscr{R}(s)$ *for the subset of states reachable from* s*.*

Definition 2.4 (Execution fragment, cf. [**BKL08**]) *Let A be an IOTS and let $s \in S_A$. A* finite execution fragment *of an IOTS A is a finite sequence $\rho = s_0 l_1 s_1 \dots l_n s_n$ of states $s_i \in S_A$ and labels $l_{i+1} \in \mathcal{L}(A)$ such that $(s_i, l_{i+1}, s_{i+1}) \in \Delta_A$ for all $0 \leq i < n$; if $n = 0$, then s_0 is a finite execution sequence. An* infinite execution fragment *of A is an infinite sequence $\rho = s_0 l_1 s_1 l_2 s_2 \dots$ of states $s_i \in S_A$ and labels $l_{i+1} \in \mathcal{L}(A)$ such that $(s_i, l_{i+1}, s_{i+1}) \in \Delta_A$ for all $i \geq 0$. An* execution fragment *of A is either a finite or an infinite execution fragment of A; with $frag^*(A)$, $frag^\omega(A)$ and $frag(A)$ we refer to the finite, infinite and to all execution fragments of A respectively. An execution fragment of A is* initial, *if it starts in the initial state $s_{0,A}$ of A. We write $frag_0^*(A)$, $frag_0^\omega(A)$ and $frag_0(A)$ for the initial respective execution fragments of A.*

We use the following notation to abbreviate conditions on the transition relation of an IOTS. Let A be an IOTS and let $\mathrm{X} \subseteq \mathcal{L}(A)$. We write $(s, s') \in \Delta_A^{\mathrm{X}}$ if either $s = s'$ or there exists $l_1, \dots, l_n \in \mathrm{X}$ and $s_1, \dots, s_{n-1} \in S_A$ such that $(s\ l_1 s_1 \dots s_{n-1} l_n\ s') \in frag(A)$; we write $(s, l, s') \in \Delta_A^{\mathrm{X}}$ if $l \notin \mathrm{X}$ and there exists $l_1, \dots, l_{k-1}, l_{k+1}, \dots l_n \in \mathrm{X}$ and $s_1, \dots, s_{n-1} \in S_A$ such that $(s\ l_1 s_1 \dots l_{k-1} s_{k-1} l\ s_{k+1} l_{k+1} \dots s_{n-1} l_n\ s') \in frag(A)$; if $X = \{\tau\}$ we write Δ_A^τ.

2.2. Operators on I/O-Transition Systems. While the definition of IOTSs and their products are motivated by interface automata [**dAH05**], hiding and relabelling are motivated by their counterparts in process algebras such as CCS [**Mil89**] or FSP [**KM99**]. A closer look reveals, however, that even the product operator strongly resembles the CCS operator for composition. Also the notions of strong and weak (observational) equivalence considered hereafter are closely related to the corresponding CCS equivalences.

We use hiding to map a subset of the labels of an IOTS to the anonymous internal action τ. Relabellings are used for renaming labels and for changing the kind of labels. The computation of the product of two IOTSs expresses their parallel composition with synchronisation on identical input and output labels. To construct the product the IOLs of the given IOTSs must be composable.

Definition 2.5 (Hiding) *The* hiding *of an IOL $L = (I, O, T)$ w.r.t. a subset $H \subseteq \bigcup L$ is the IOL $L/H = (I \setminus H, O \setminus H, T \setminus H)$. The* hiding *of an IOTS $A = (L, S, s_0, \Delta)$ w.r.t. a label set $H \subseteq \bigcup L$ is the IOTS $A/H = (L/H, S, s_0, \Delta/H)$ where $\Delta/H = \{(s, \tau, s') \mid (s, l, s') \in \Delta \wedge l \in H\} \cup \{(s, l, s') \mid (s, l, s') \in \Delta \wedge l \notin H\}$.*

In many cases we will choose $H = T$, i.e., we will hide all internal labels. Then, for an IOL $L = (I, O, T)$, we write $L\xi$ for L/T and, for an IOTS A, we write $A\xi$ for A/T_A.

Definition 2.6 (Relabelling) *A* relabelling *$\rho : L \to L'$ from an IOL $L = (I, O, T)$ to an IOL $L' = (I', O', T')$ is defined by a function f_ρ from $\bigcup L$ to $\bigcup L'$. The* relabelling *of an IOTS $A = (L, S, s_0, \Delta)$ w.r.t. a relabelling $\rho : L \to L'$ is the IOTS $A\rho = (L', S, s_0, \Delta\rho)$ where $\Delta\rho = \{(s, f_\rho(l), s') \mid (s, l, s') \in \Delta \wedge l \in \bigcup L\} \cup \{(s, \tau, s') \mid (s, \tau, s') \in \Delta\}$.*

For convenience we usually use the same name for a relabelling and its defining function, i.e. use ρ instead of f_ρ. A particular application of relabelling is the internalisation of (some) input and output labels. For an IOL $L = (I, O, T)$ and a subset $X \subseteq I \cup O$, we define the *internalisation relabelling* $\theta_X : (I, O, T) \to (I \setminus X, O \setminus X, T \cup X)$ with $\theta_X(l) = l$ for $l \in \bigcup L$. Moreover, we need to compute the union of two relabellings for the same IOL. Given two relabellings ρ_1 and ρ_2 which coincide on all elements in the intersection of their domains, we denote their union (as function graphs) by $\rho_1 \cup \rho_2$.

Definition 2.7 (Composability and shared labels) *Let $L_1 = (I_1, O_1, T_1)$ and $L_2 = (I_2, O_2, T_2)$ be IOLs. (L_1, L_2) are* composable *if $I_1 \cap I_2 = \emptyset$, $O_1 \cap O_2 = \emptyset$, $T_1 \cap (I_2 \cup O_2 \cup T_2) = \emptyset$, and $T_2 \cap (I_1 \cup O_1 \cup T_1) = \emptyset$. Two IOTSs A_1 and A_2 are* composable *if L_{A_1} and L_{A_2} are composable. The* shared *labels of composable IOLs $L_1 = (I_1, O_1, T_1)$ and $L_2 = (I_2, O_2, T_2)$, written $L_1 \bowtie L_2$, are given by $(I_1 \cap O_2) \cup (O_1 \cap I_2)$. Let*

$$\Delta = \{((s_1, s_2), l, (s_1', s_2)) \mid (s_1, l, s_1') \in \Delta_1 \wedge s_2 \in S_2 \wedge l \notin L_1 \bowtie L_2\} \cup$$
$$\{((s_1, s_2), l, (s_1, s_2')) \mid (s_2, l, s_2') \in \Delta_2 \wedge s_1 \in S_1 \wedge l \notin L_1 \bowtie L_2\} \cup$$
$$\{((s_1, s_2), l, (s_1', s_2')) \mid (s_1, l, s_1') \in \Delta_1 \wedge (s_2, l, s_2') \in \Delta_2 \wedge l \in L_1 \bowtie L_2\} \,.$$

FIGURE 2.5. Transition relation for IOTS products

$A_1, \dots, A_n$ be pairwise composable IOTSs. The set of shared labels of $A_1, \dots, A_n$ is given by $\mathcal{S}(A_1, \dots, A_n) = \bigcup_{i \neq j} L(A_i) \bowtie L(A_j)$.

Two partitioned IOLs are composable, if their underlying label sets are composable and shared labels can be uniquely assigned to the partitions.

Definition 2.8 (Composability of partitioned I/O-labellings) *Let $L_A = ((I_{A,1}, O_{A,1}), \dots, (I_{A,n}, O_{A,n}), T_A)$ and $L_B = ((I_{B,1}, O_{B,1}), \dots, (I_{B,m}, O_{B,m}), T_B)$ be partitioned IOLs. L_A and L_B are composable if $(\bigcup_{i=1}^{n}(I_{A,i}), \bigcup_{i=1}^{n}(O_{A,i}), T_A)$ and $(\bigcup_{i=1}^{m}(I_{B,i}), \bigcup_{i=1}^{n}(O_{B,i}), T_B)$ are composable, and if $l \in \mathcal{S}(A, B)$, then there exists unique $i, j \in \mathbb{N}$ such that $l \in (I_{A,i} \cap O_{B,j}) \cup (O_{A,i} \cap I_{B,j})$.*

Definition 2.9 (Composition) *The* composition (product) *of two composable IOLs L_1 and L_2 is the IOL $L_1 \otimes L_2 = ((I_1 \cup I_2) \setminus (L_1 \bowtie L_2), (O_1 \cup O_2) \setminus (L_1 \bowtie L_2), T_1 \cup T_2 \cup (L_1 \bowtie L_2))$. The* product *of two composable IOTSs $A_1 = (L_1, S_1, s_{0,1}, \Delta_1)$ and $A_2 = (L_2, S_2, s_{0,2}, \Delta_2)$ is the IOTS $A_1 \otimes A_2 = (L_1 \otimes L_2, S_1 \times S_2, (s_{0,1}, s_{0,2}), \Delta)$ where Δ is given by Tab. 2.5*

Pairwise composability for a set of IOLs implies that all composable pairs of this set have mutually disjoint shared labels. In the context of IOTS products, mutually disjoint shared labels guarantee that the synchronisation between different IOTS is always binary which is a prerequisite for the associativity of the IOTS product operator (cf. composability of interface automata [**dAH01b**, Theorem 3.1] or [**EDM08**, Theorem 2.1, 2.2]).

Lemma 2.10 *If the IOLs $L_1, \dots, L_n$ are pairwise composable, then for all $i \neq j \in \{1, \dots, n\}$, $(L_i, L_j) \neq (L_i', L_j') \in \{(L, L') \mid L \neq L' \in \{L_1, \dots, L_n\} \wedge L \bowtie L' \neq \emptyset\}$ it holds that $(L_1 \bowtie L_2) \cap (L_1' \bowtie L_2') = \emptyset$.*

PROOF. Pairwise composability means that for all $i, j \in \{1, \dots, n\}$ with $i \neq j$ and $L_i = (I_i, O_i, T_i)$ and $L_j = (I_j, O_j, T_j)$ it holds that $I_i \cap I_j = \emptyset$, $O_i \cap O_j = \emptyset$ and $T_i \cap \bigcup L_j = \emptyset$. Assume there exists $(L_1, L_2) \neq (L_1', L_2')$ such that $(L_1 \bowtie L_2) \cap (L_1' \bowtie L_2') \neq \emptyset$. Then, by definition of IOL products, $((I_1 \cap O_2) \cup (I_2 \cap O_1)) \cap ((I_1' \cap O_2') \cup (I_2' \cap O_1') \neq \emptyset$. W.l.o.g. assume that $(I_1 \cap O_2) \cap (I_1' \cap O_2') \neq \emptyset$, then $I_1 \cap I_1' \cap O_2 \cap O_2' \neq \emptyset$, contradicting pairwise compatibility which requires $I_1 \cap I_1' = \emptyset$ and $O_2 \cap O_2' = \emptyset$. □

Composability is preserved by IOL subsets. For $L = (I, O, T)$, $L' = (I', O', T')$ IOLs, we write $L' \subseteq L$ iff $I' \subseteq I$, $O' \subseteq O$ and $T' \subseteq T$.

Lemma 2.11 *Let L_1, L_2 and $L_2' \subseteq L_2$ be IOLs. If L_1 and L_2 are composable, then L_1 and L_2' are composable.*

PROOF. For $i = 1, 2$, let $L_i = (I_i, O_i, T_i)$ be composable and let $L_2' = (I_2', O_2', T_2') \subseteq L_2$. Then $I_1 \cap I_2 = \emptyset$ implies $I_1 \cap I_2' = \emptyset$, $O_1 \cap O_2 = \emptyset$ implies $O_1 \cap O_2' = \emptyset$, $T_1 \cap \bigcup L_2 = \emptyset$ implies $T_1 \cap \bigcup L_2' = \emptyset$ and $T_2 \cap \bigcup L_1 = \emptyset$ implies $T_2' \cap \bigcup L_1 = \emptyset$. □

2.3. Greybox and Blackbox Equivalence. We define three kinds of equivalence relations between IOTSs. First, strong equivalence which can be considered to be a substitute for equality. Second, greybox equivalence which abstracts from τ and is useful for example for the minimisation of assembly behaviours in case internal synchronisation transitions should not be hidden away. And third, blackbox (observational) equivalence which makes no difference between internal and τ transitions and abstracts from both.

Definition 2.12 (Strong equivalence) *A* strong bisimulation *between two IOTSs A and B with the same IOL $L_A = L_B = L$ is a relation $R \subseteq S_A \times S_B$ such that for all $(s_A, s_B) \in R$ and all $l \in \bigcup L \cup \{\tau\}$ the following holds:*

$$\forall s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta_A \implies \exists s'_B \in S_B \,.\, (s_B, l, s'_B) \in \Delta_B \wedge (s'_A, s'_B) \in R,$$
$$\forall s'_B \in S_B \,.\, (s_B, l, s'_B) \in \Delta_B \implies \exists s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta_A \wedge (s'_A, s'_B) \in R.$$

The IOTSs A and B are strongly equivalent*, denoted by $A \approx^{\mathrm{s}} B$, if there exists a strong bisimulation R between A and B with $(s_{0,A}, s_{0,B}) \in R$.*

Definition 2.13 (Greybox equivalence) *A* gb-bisimulation *between two IOTSs A and B with the same IOL $L_A = L_B = L$ is a relation $R \subseteq S_A \times S_B$ such that for all $(s_A, s_B) \in R$ and all $l \in \bigcup L$ the following holds:*

$$\forall s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta_A \implies \exists s'_B \in S_B \,.\, (s_B, l, s'_B) \in \Delta^{\tau}_B \wedge (s'_A, s'_B) \in R,$$
$$\forall s'_A \in S_A \,.\, (s_A, \tau, s'_A) \in \Delta_A \implies \exists s'_B \in S_B \,.\, (s_B, s'_B) \in \Delta^{\tau}_B \wedge (s'_A, s'_B) \in R,$$
$$\forall s'_B \in S_B \,.\, (s_B, l, s'_B) \in \Delta_B \implies \exists s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta^{\tau}_A \wedge (s'_A, s'_B) \in R,$$
$$\forall s'_B \in S_B \,.\, (s_B, \tau, s'_B) \in \Delta_B \implies \exists s'_A \in S_A \,.\, (s_A, s'_A) \in \Delta^{\tau}_A \wedge (s'_A, s'_B) \in R.$$

The IOTSs A and B are greybox equivalent*, denoted by $A \approx^{\mathrm{gb}} B$, if there exists a gb-bisimulation R between A and B with $(s_{0,A}, s_{0,B}) \in R$.*

Definition 2.14 (Blackbox equivalence) *A* bb-bisimulation *between two IOTSs A and B with the same IOL (I, O, T) is a relation $R \subseteq S_A \times S_B$ such that for all $(s_A, s_B) \in R$ and all $l \in I \cup O$ and all $t \in T \cup \{\tau\}$ the following holds:*

$$\forall s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta_A \implies \exists s'_B \in S_B \,.\, (s_B, l, s'_B) \in \Delta^{T \cup \{\tau\}}_B \wedge (s'_A, s'_B) \in R,$$
$$\forall s'_A \in S_A \,.\, (s_A, t, s'_A) \in \Delta_A \implies \exists s'_B \in S_B \,.\, (s_B, s'_B) \in \Delta^{T \cup \{\tau\}}_B \wedge (s'_A, s'_B) \in R,$$
$$\forall s'_B \in S_B \,.\, (s_B, l, s'_B) \in \Delta_B \implies \exists s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta^{T \cup \{\tau\}}_A \wedge (s'_A, s'_B) \in R,$$
$$\forall s'_B \in S_B \,.\, (s_B, t, s'_B) \in \Delta_B \implies \exists s'_A \in S_A \,.\, (s_A, s'_A) \in \Delta^{T \cup \{\tau\}}_A \wedge (s'_A, s'_B) \in R.$$

The IOTSs A and B are blackbox equivalent*, denoted by $A \approx^{\mathrm{bb}} B$, if there exists a bb-bisimulation R between A and B with $(s_{0,A}, s_{0,B}) \in R$.*

Lemma 2.15 *Let A and B be IOTSs with the same IOL. Then $A \approx^{\mathrm{bb}} B$ if and only if $A\xi \approx^{\mathrm{gb}} B\xi$.* □

In the following we list basic properties for greybox and blackbox equivalence that are used mainly in Chap. 3 of this thesis. As the following claims hold analogously for blackbox equivalence, we use the notation $\approx$ to denote either greybox or blackbox equivalence.

Greybox (blackbox) equivalence is a congruence w.r.t. hidings, relabellings and products. For hiding and relabelling this follows directly from the definitions and corresponds to the respective facts for standard process algebras, like [**KM99**], since greybox (blackbox) equivalence is independent of the kind of the labels. The preservation of greybox (blackbox) equivalence by the product follows from the facts that greybox (blackbox) equivalence is a congruence w.r.t. parallel composition with synchronisation of shared labels [**KM99**] and that the equality of labellings is preserved by the product.

Lemma 2.16 *Let A, B, and C be IOTSs.*

(1) If $A \approx B$ and $H \subseteq \bigcup L_A$, then $A/H \approx B/H$.
(2) If $A \approx B$ and $\rho : L_A \to L'$ is a relabelling, then $A\rho \approx B\rho$.
(3) If $A \approx B$ and A and C are composable, then $A \otimes C \approx B \otimes C$. □

The product is commutative and associative up to greybox (blackbox) equivalence. Commutativity is straightforward from the definitions; associativity carries over from the corresponding fact for interface automata [**dAH05**] taking into account also τ and internal

transitions. Furthermore, hiding of non-shared labels commutes with the formation of products.

Lemma 2.17 *Let A, B, and C be IOTSs.*

(1) If A and B are composable, then $A \otimes B \approx B \otimes A$.
(2) If A, B, and C are pairwise composable, then $(A \otimes B) \otimes C \approx A \otimes (B \otimes C)$.
(3) If A and B are composable and $H \subseteq \bigcup L_A$ with $H \cap \mathcal{S}(A, B) = \emptyset$, then $(A/H) \otimes B \approx (A \otimes B)/H$. □

For a finite index set I, we write $\bigotimes_{i \in I} A_i$ for the product of the IOTSs A_i with $i \in I$, which is justified by Lem. 2.17 (2).

2.4. Encoding of FIFO Queues. Asynchronous communication is supported within our component model in the form of buffered communication using FIFO queues. We encode FIFO queues by IOTSs over a given set M of messages such that storing a message m in the queue is an input action of the form $m^{\triangleright}$ and removing a message m from the queue is an output action of the form m. The contents of a queue define the queue's states which are formally represented by the set M^* of finite sequences over M. The empty sequence is denoted by ϵ, the extension of $s \in M^*$ by an $m \in M$ at the head of the queue is written $\langle ms \rangle$, and extension at the tail is written $\langle sm \rangle$.

Definition 2.18 (Output queue) *Let M be a set. An* output queue *over M is given by $Q^{\triangleright}_M = ((I, O, T), S, s_0, \Delta)$, where $I = \{m^{\triangleright} \mid m \in M\}$, $O = \{m \mid m \in M\}$, $T = \emptyset$; $S = \{s \mid s \in M^*\}$, $s_0 = \epsilon$; and $\Delta \subseteq S \times (I \cup O \cup T \cup \{\tau\}) \times S$ is the smallest relation such that*

(1) for all $m^{\triangleright} \in I$ and all $s \in S$, there exists $(s, m^{\triangleright}, \langle sm \rangle) \in \Delta$,
(2) for all $m \in O$ and all $\langle ms \rangle \in S$, there exists $(\langle ms \rangle, m, s) \in \Delta$.

Output queues will be used to formalise the buffering behaviour of asynchronous connectors. For synchronous connectors we apply a notion of "empty IOTS" since synchronous communication is already completely covered by the properties of the IOTS product operator. An empty IOTS is given by $\mathbf{0} = (L, S, s_0, \Delta)$ with $\bigcup L = \emptyset$, $S = \{s_0\}$ and $\Delta = \emptyset$. By queue and empty IOTSs we obtain a uniform definition of buffering behaviours of connectors which is needed for the definition of assembly behaviour below (Def. 2.22).

3. Components with (A)Synchronous Communication

In this section we make the concepts introduced above using a metamodel more precise. For the formalisation of behaviours and asynchronous connectors we will use IOTSs with the given encoding of FIFO queues.

3.1. Ports, Components, and Connectors. For the formalisation of ports we assume a domain Port of ports (more precisely, port types), a domain If of interfaces and a domain Msg of messages, together with functions $msg : \mathrm{If} \to \wp\mathrm{Msg}$ to return the messages constructed from the operations of an interface, and $prv : \mathrm{Port} \to \mathrm{If}$ and $req : \mathrm{Port} \to \mathrm{If}$ for the provided and required interfaces of a port such that for all $P \in \mathrm{Port}, msg(prv(P)) \cap msg(req(P)) = \emptyset$. For a port P we write $msg(P)$ for $msg(prv(P)) \cup msg(req(P))$. We also assume a domain of port declarations $\mathrm{PortDcl}$ with a function $nm : \mathrm{PortDcl} \to \mathrm{Nm}$ for the name and a function $ty : \mathrm{PortDcl} \to \mathrm{Port}$ for the port (type); we write $p : P$ for a port declaration d with $nm(d) = p$ and $ty(d) = P$.

If ports are used for asynchronously communicating components, an output queue is defined with respect to the messages required at the particular port.

Definition 2.19 (Port queue) *The* queue *of a port (type) P is given by*

$$que(P) = Q^{\triangleright}_{msg(req(P))} \ .$$

We assume a domain Cmp of components (more precisely, component types) and a function $ports : \mathrm{Cmp} \to \wp\mathrm{PortDcl}$ returning the ports declared for a component. For a component C and port declaration $p : P$ we write $C[p : P]$ to indicate that $p : P \in ports(C)$. The port names p used in port declarations $p : P$ of one component C must be unique (but this is not necessary for port types P). Like for ports and connectors, we assume a domain of component declarations CmpDcl with a function $nm : \mathrm{CmpDcl} \to \mathrm{Nm}$ for the name and a function $ty : \mathrm{CmpDcl} \to \mathrm{Cmp}$ for the component (type); we write $c : C$ for a component declaration d with $nm(d) = c$ and $ty(d) = C$. The ports of a component declaration are given by $ports(c : C) = \{c.p : P \mid p : P \in ports(C)\}$. We assume a sub-domain $\mathrm{SCmp} \subseteq \mathrm{Cmp}$ of simple components. Each $SC \in \mathrm{SCmp}$ has a user defined behaviour specification.

Definition 2.20 (Behaviour simple component) *The* behaviour *of a simple component SC is given by an IOTS*

$$beh(SC) = ((I, O, T), S, s_0, \Delta) ,$$

where $I = \{p.msg(prv(P)) \mid p : P \in ports(SC)\}$ and $O = \{p.msg(req(P)) \mid p : P \in ports(SC)\}$.

The labels of a component behaviour are just the labels according to the (provided and required) messages of the ports of the component prefixed with the port name in the corresponding port declaration. Additional actions that may occur are either labelled by distinguished internal labels or by the anonymous internal action τ. For the construction of assemblies component declarations are used. Then the behaviour of a component should be uniquely represented within the global behaviour and hence we define the *behaviour of a component declaration* $c : C \in \mathrm{CmpDcl}$ by $beh(c : C) = c.beh(C)$ using a prefix relabelling of the original behaviour. A prefix relabelling prefixes all labels of an IOTS by some given name. We assume a primitive domain Nm of *names*. For an IOL $L = (I, O, T)$ and some name $n \in \mathrm{Nm}$, we define the IOL $n.L = (n.I, n.O, n.T)$ where $n.I = \{n.i \mid i \in I\}$ and similarly for $n.O$ and $n.T$. The *prefix relabelling* $\rho_n : L \to n.L$ is defined by $\rho_n(l) = n.l$ for $l \in \bigcup L$. Given an IOTS A and a name $n \in \mathrm{Nm}$, we write $n.A$ for the IOTS $A\rho_n$.

For building component assemblies and composite components we will connect port declarations of components. We assume a domain Conn of connectors (more precisely, connector types) with a function $ports : \mathrm{Conn} \to \wp\mathrm{PortDcl}$ yielding the connected port declarations such that $1 \leq |ports(K)| \leq 2$ for each $K \in \mathrm{Conn}$, i.e. we consider unary and binary connectors. We assume a domain of connector declarations $\mathrm{ConnDcl}$ with a function $nm : \mathrm{ConnDcl} \to \mathrm{Nm}$ for the name and $ty : \mathrm{ConnDcl} \to \mathrm{Conn}$ for the connector (type); we write $k : K$ for a connector declaration d with $nm(d) = k$ and $ty(d) = K$ and we write $k : (p : P, q : Q)$ if $ports(K) = \{p : P, q : Q\}$.

The domain Conn has two disjoint sub-domains $\mathrm{AsmConn} \subseteq \mathrm{Conn}$, $\mathrm{DlgConn} \subseteq \mathrm{Conn}$ of assembly and delegate connectors, resp. Assembly connectors are used to connect port declarations of components when building up a component assembly. For an assembly connector with port declarations $\{p_1 : P_1, p_2 : P_2\}$ the required interface $req(P_1)$ has to be equal to the provided interface $prv(P_2)$ and vice versa. There are again two disjoint sub-domains $\mathrm{AsynConn} \subseteq \mathrm{AsmConn}$, $\mathrm{SynConn} \subseteq \mathrm{Conn}$ for asynchronous and synchronous connectors, resp.

Delegate connectors are used to connect open ports of an assembly with the relay ports of a surrounding composite component. For a delegate connector the provided and required interfaces of its port declarations must coincide.

Asynchronous connectors are used for asynchronous communication between the ports of components. They show a buffering behaviour which relies on the queues defined for the connected ports.

Definition 2.21 (Buffering connector behaviour) *The* buffering behaviour *of an asynchronous connector* $k : (p : P, q : Q)$ *is given by*

$$buf(k : (p : P, q : Q)) = k.(que(P) \otimes que(Q)) \ .$$

If a connector $k : K$ *is synchronous we set* $buf(k : K) = \mathbf{0}$.

3.2. Assemblies and Composite Components. For the computation of the behaviour of assemblies and composite components we employ several relabelling functions, which are fixed cases of IOTS relabellings as introduced in Sect. 2. These relabellings are needed for creating shared labels when I/O-transition systems, representing behaviours, are composed. Even for asynchronous compositions shared labels will be needed for appropriate synchronisations with the queue transition systems.

A match relabelling maps differently prefixed labels to labels with a single common prefix. For an IOL $L = (I, O, T)$, $X \subseteq \mathrm{Nm}$ and $y \in \mathrm{Nm}$, we define the IOL $L\mu_{(X,y)} = (I\mu_{(X,y)}, O\mu_{(X,y)}, T\mu_{(X,y)})$ where $I\mu_{(X,y)} = \{y.l \mid \exists x \in X \,.\, x.l \in I\} \cup \{l \mid l \in I \wedge \forall x \in X \,.\, l \neq x.l'\}$ and analogously for $O\mu_{(X,y)}$ and $T\mu_{(X,y)}$. The *match relabelling* $\mu_{(X,y)} : L \to L\mu_{(X,y)}$ is defined by $\mu_{(X,y)}(x.l) = y.l$ if $x \in X$ and $x.l \in \bigcup L$, and $\mu_{(X,y)}(l') = l'$ otherwise.

A *unary synchronisation relabelling* $\sigma_{(x,y)}$ is given by the composition $\theta_X \circ \mu_{(\{x\},y)}$ of a match relabelling and an internalisation with $X = \{y.l \mid y.l \in I\mu_{(\{x\},y)} \cup O\mu_{(\{x\},y)}\}$. For an IOL $L = (I, O, T)$, $X \subseteq \mathrm{Nm}$ and $y \in \mathrm{Nm}$, a (binary) *synchronisation relabelling* $\sigma_{(X,y)}$ is given by a match relabelling $\mu_{(X,y)}$ with $|X| = 2$ and $T = T\mu_{(X,y)}$.

An *asynchronous relabelling* $\alpha_{(X,y)}$ is given by the composition $\sigma_{(X,y)} \circ \beta_U$ with $U = \{x.l \in I \cup O \mid x \in X, l \in \mathrm{Nm}\}$ and β_U a relabelling defined as follows: For an IOL (I, O, T) and a set of labels M define $O_{M^\triangleright} = \{l^\triangleright \mid l \in O \cap M\} \cup (O \setminus M)$; now $\beta_U : (I, O, T) \to (I, O_{M^\triangleright}, T)$ is defined by $\beta_U(l) = l^\triangleright$ if $l \in O \cap U$, and $\beta_U(l) = l$ otherwise. Finally, a *relay relabelling* $\rho_{(x,y)}$ is given by a match relabelling $\mu_{(X,y)}$ with $X = \{x\}$.

An assembly contains a set of component declarations and a set of connector declarations which connect ports (more precisely, the connector declarations connect port declarations belonging to component declarations of the assembly). We assume a domain Asm of assemblies with functions $cmps : \mathrm{Asm} \to \wp\mathrm{CmpDcl}$ returning an assembly's declared components and $conns : \mathrm{Asm} \to \wp\mathrm{ConnDcl}$ yielding its declared connectors. The component names c used in component declarations $c : C$ of an assembly a must be unique (but this is not necessary for the component types C). Similarly, connector names within the assembly must be unique. For an assembly a we define the subset of asynchronous connectors by $acs(a) \subseteq conns(a)$ such that $k : K \in acs(a)$ iff $K \in \mathrm{AsynConn}$, and the subset of synchronous connectors by $scs(a) \subseteq conns(a)$ such that $k : K \in scs(a)$ iff $K \in \mathrm{SynConn}$. The remaining connectors in $conns(a)$ are unary and we define $ucs(a) = conns(a) \setminus (acs(a) \cup scs(a))$.

An assembly $a \in \mathrm{Asm}$ has to be well-formed: (i) it shows only assembly connectors, i.e., if $k : K \in conns(a)$, then $K \in \mathrm{AsmConn}$; (ii) only ports of components inside a are connected, i.e., for all $k : K \in conns(a)$ we have that $ports(K) \subseteq \bigcup\{ports(c : C) \mid c : C \in cmps(a)\}$; and (iii) there is at most one connector for each port, i.e., if $c.p : P \in \bigcup\{ports(c : C) \mid c : C \in cmps(a)\}$ and $k : K, k' : K' \in conns(a)$ with $c.p : P \in ports(K) \cap ports(K')$, then $k : K = k' : K'$.

To retrieve component declarations from port declarations within an assembly a we define $cmp : \bigcup\{ports(c : C) \mid c : C \in cmps(a)\} \to cmps(a)$ by $cmp(c.p : P) = c : C$ if $c.p : P \in ports(c : C)$. The components of an assembly a may show *open* ports which are not connected and we let $open(a) = \bigcup\{ports(c : C) \mid c : C \in cmps(a)\} \setminus \bigcup\{ports(K) \mid k : K \in conns(a)\}$.

We write $\langle \mathcal{C}; \mathcal{K} \rangle$ for an assembly a with the set of component declarations $cmps(a) = \mathcal{C}$ and the set of connector declarations $conns(a) = \mathcal{K}$. We will use a notation to substitute

(replace) components within a given assembly. More precisely, we need to replace the component type of some component declaration given by an assembly. Let $a = \langle \mathcal{C}; \mathcal{K} \rangle$ be an assembly, $c : C \in \mathcal{C}$ and $C' \in \mathrm{Cmp}$ with $ports(C') = ports(C)$. The *(type) substitution of* $c : C$ *by* C' *in* a, written $a[c : C \mapsto C']$, is given by $\langle (\mathcal{C} \setminus \{c : C\}) \cup \{c : C'\}; \mathcal{K} \rangle$. We allow also for an n-ary simultaneous substitution written $a[c_1 : C_1 \mapsto C'_1, \ldots, c_n : C_n \mapsto C'_n]$.

In the following we focus on the definition of the behaviour of an assembly. The idea is that the behaviour of an assembly is determined by the composition of the behaviours of the components occurring in the assembly. But, of course, the composition must be defined in accordance with the possible communications between components which are connected via their ports. Since connectors may be asynchronous the buffering behaviour of connectors (cf. Def. 2.21) plays a crucial role. Moreover, match relabellings are necessary to ensure proper synchronisation.

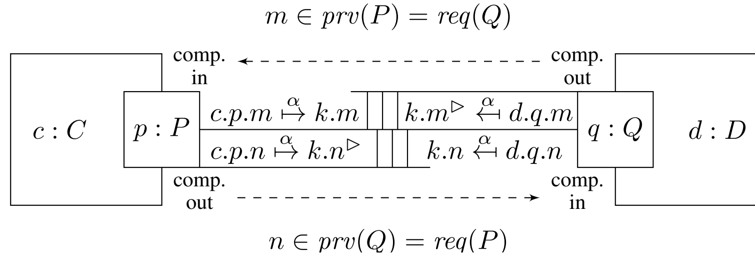


FIGURE 2.6. Assembly with asynchronous connector

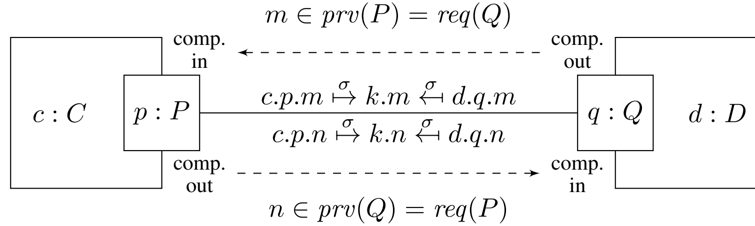


FIGURE 2.7. Assembly with synchronous connector

Figure 2.6 illustrates how the behaviour of an assembly with two asynchronously communicating components is constructed. There are two component declarations $c : C$ and $d : D$. The component type C has one port declaration $p : P$ which, in the context of the assembly with component declaration $c : C$, is considered as a port declaration $c.p : P$ to ensure uniqueness of port names within an assembly. Similarly, the component type D has one port declaration $q : Q$. Thus the messages sent out from the component c via its port p have the form $c.p.n$ where n is a message of the required interface of P. The two ports are connected by a connector declaration of the form $k : (c.p : P, d.q : Q)$. Thus the required interface of P must coincide with the provided interface of Q. According to the buffering behaviour of the connector k there is a queue $que(P)$ (cf. Def. 2.19) which allows to store outputs of the form $c.p.n$ with n being a message according to the required interface of P. To achieve that the issued message $c.p.n$ will indeed be put into the queue $que(P)$, we use an asynchronous relabelling α which maps $c.p.n$ to $k.n^{\triangleright}$. The message n can be dequeued from $que(P)$ later on with the action $k.n$. Since $d.q.n$ is an input of component d on port q, only the synchronous part of the matching relabelling α has an effect and maps $d.q.n$ to $k.n$. The communication in the other direction works analogously.

Figure 2.7 illustrates how the behaviour of an assembly with two synchronously communicating components is constructed. Here, the necessary relabelling to synchronise input and output actions is much easier. Indeed, in this case a message $c.p.n$ sent from component c via its port p must be matched with the input action $d.q.n$ on the port q of the component d. For this purpose both actions are simply matched to the label $k.n$ with the relabelling σ, where k is again the connector's name.

For general assemblies a, we denote the unary and binary synchronisation relabellings by σ and the asynchronous relabellings by α as follows:

$$\begin{aligned}\sigma \equiv & \bigcup\{\sigma_{(c.p,k)} \mid k : K \in \mathit{ucs}(a) \,.\, \mathit{ports}(K) = \{c.p : P\}\} \cup \\ & \bigcup\{\sigma_{(\{c.p,d.q\},k)} \mid k : K \in \mathit{scs}(a) \,.\, \mathit{ports}(K) = \{c.p : P, d.q : Q\}\}, \\ \alpha \equiv & \bigcup\{\alpha_{(\{c.p,d.q\},k)} \mid k : K \in \mathit{acs}(a) \,.\, \mathit{ports}(K) = \{c.p : P, d.q : Q\}\}.\end{aligned}$$

We have now the technical ingredients to define the behaviour of an assembly. In the definition the successive application of σ and α cannot lead to conflicts because both relabellings are disjoint due to the well-formedness of assemblies. Note also that in the case of synchronous connectors $\mathit{buf}(k : K)$ is trivial as explained in Sect. 3.1. Moreover, the assembly behaviour is well-defined because all participating behaviours are composable. This is due to the disjointness of provided and required operations on port types, to the uniqueness of names for ports (in component declarations) as well as for components and connectors (in the assembly), and to the commutativity and associativity of the composition operator for IOTSs.

Definition 2.22 (Assembly behaviour) *The* behaviour *of an assembly a is given by*

$$\mathit{beh}(a) = \bigotimes_{c:C\in \mathit{cmps}(a)}((\mathit{beh}(c : C)\xi)\sigma\alpha) \otimes \bigotimes_{k:K\in \mathit{conns}(a)} \mathit{buf}(k : K)\ .$$

We apply the ξ operator, hiding local internal actions of the components before composition and obtain an assembly behaviour that shows a greybox view with synchronisation labels of the composed components and connectors. The hiding of local internal actions corresponds to the hiding of implementation details which are not relevant for the given hierarchical level any more. Indeed relevant, for instance for the verification of communication properties, are synchronisation transitions of components, and of components with message buffers respectively. We take this into account by a corresponding greybox view that leaves the communication via synchronous and asynchronous connectors internally visible.

Composite components are constructed by encapsulating an assembly and by connecting, with delegate connectors, the open ports of the assembly with relay ports of the composite component. Formally, we assume a sub-domain $\mathrm{CCmp} \subseteq \mathrm{Cmp}$ of composite components, disjoint to SCmp and functions $\mathit{asm} : \mathrm{CCmp} \to \mathrm{Asm}$ returning the underlying assembly of a composite component, and $\mathit{conns} : \mathrm{CCmp} \to \wp\mathrm{ConnDcl}$ returning the connectors declared in a composite component. Similar to assemblies we require a composite component CC to be well-formed: (i) it shows only delegate connectors, i.e., if $k : K \in \mathit{conns}(CC)$, then $K \in \mathrm{DlgConn}$; (ii) all open ports of the $\mathit{asm}(CC)$ are connected, i.e., for all $c.p : P \in \mathit{open}(\mathit{asm}(CC))$ there is $k : K \in \mathit{conns}(CC)$ such that $c.p : P \in \mathit{ports}(K)$; and (iii) all relay ports are connected, i.e., for all $r : R \in \mathit{ports}(CC)$ there is a unique $k : K \in \mathit{conns}(CC)$ with $\mathit{ports}(K) = \{c.p : P, r : R\}$ and $c.p : P \in \mathit{open}(\mathit{asm}(CC))$.

We write $\langle a; \mathcal{P}; \mathcal{K}\rangle$ for a composite component CC with assembly $\mathit{asm}(CC) = a$, set of (relay) port declarations $\mathit{ports}(CC) = \mathcal{P}$ and set of (delegate) connector declarations $\mathit{conns}(CC) = \mathcal{K}$. Finally, we extend the notion of type substitution for assemblies to composite components. Let $CC = \langle a; \mathcal{P}; \mathcal{D}\rangle$ a composite component, $c : C \in \mathit{cmps}(a)$ and $C' \in \mathrm{Cmp}$ with $\mathit{ports}(C') = \mathit{ports}(C)$. The *(type) substitution of* $c : C$ *by* C' *in* CC, written $CC[c : C \mapsto C']$, is given by $\langle a[c : C \mapsto C']; \mathcal{P}; \mathcal{D}\rangle$.We allow also for an n-ary simultaneous substitution written $CC[c_1 : C_1 \mapsto C'_1, \ldots, c_n : C_n \mapsto C'_n]$.

The behaviour of a composite component is derived from the behaviour of its underlying assembly by matching the labels on the open ports of the assemblies with the labels on the relay ports in accordance with the delegate connectors.

Definition 2.23 (Behaviour of composite component) *The* behaviour *of a composite component CC is given by*

$$beh(CC) = beh(asm(CC))\rho \ ,$$

where $\rho = \bigcup\{\rho_{(c.p,r)} \mid k : K \in conns(CC) \,.\, ports(K) = \{c.p : P, r : R\}\}$.

The behaviour of composite components shows internal transitions that arise from the synchronisation of components and connectors within the underlying assembly. Thus the behaviour is a greybox behaviour analogous to the case of simple components. If a composite component is set into the context of a new assembly then the blackbox view of the component's behaviour is used and all implementation details of the composite component are relabelled to anonymous internal τ actions.

4. Implementation using Message-Oriented Middleware

Port-based component systems can be implemented using a message-oriented middleware (MOM) such as *Open Message Queue* [**Mic07**, Open MQ] which is an implementation of the *Java Message Service* standard [**Mic02**, JMS]. A MOM provides a framework for the realisation of distributed systems whose components communicate by asynchronous message exchange. Components are loosely coupled without means for direct interaction via method calls, remote method calls respectively. Data is transfered between different components as part of messages only. Thus the basic communication mechanism is sending and receiving of messages using a layer that provides data structures and mechanisms to construct, buffer and deliver messages. Figures 1.2 and 1.4 show schematic views on the role of the middleware layer as part of the implementation of distributed systems.

4.1. Java Message Service and Open MQ. JMS is a specification, provided by Sun Microsystems, that defines the basic entities of a message oriented system and how to access the MOM layer in order to communicate messages. The specification comprises Java interfaces with informal descriptions that can be used by Java programs to access a MOM that implements the JMS standard. As an example for such an implementation we consider Open MQ which is a reference implementation of JMS also provided by Sun. A Java program that uses JMS is called a *JMS application*.

The JMS standard essentially consists of four parts. The *architecture* describes what constitutes a JMS application, the *message model* defines the detailed structure of JMS messages, the *communication model* describes the two basic messaging styles in JMS applications, point-to-point (P2P) and publish-subscribe messaging, and finally *common facilities* comprise concepts that are common to these styles such as connections or sessions for sending and receiving messages. After a brief architectural overview, we will focus on concepts for JMS applications with P2P messaging. These concepts provide an appropriate foundation for the implementation of port-based components with asynchronous communication. Publish-subscribe messaging can be used to realise group communication. Though, group communication was not considered within this thesis and hence publish-subscribe messaging does not play a particular role in the following.

A JMS application comprises a *JMS Provider* and essentially three kinds of entities: *JMS Clients*, *Messages*, and *Administered Objects*. The JMS provider is the complete messaging system that implements the JMS API as well as the administrative and control functionality of the middleware. JMS Clients are Java programs that create, send and receive Messages. Administered objects are JMS objects that are implemented and managed by the JMS Provider. For instance, the queues used for buffered communication are administered objects.

JMS specifies, amongst others the following interfaces (cf. [**Mic02**, Tab. 2-1]), relevant for an implementation of port-based components with asynchronous communication:

`QueueConnectionFactory`	to create connections with JMS Provider
`QueueConnection`	to create sessions
`Queue`	administered object for message exchange
`QueueSession`	to create sender and receiver for each queue
`QueueSender`	queue specific object to send messages
`QueueReceiver`	queue specific object for message reception
`MessageListener`	for asynchronous message reception

The basic procedure to set up a JMS application with P2P communication comprises the following steps. First, use the `QueueConnectionFactory` to get a connection to the JMS Provider (middleware). Next, use the `QueueConnection` to create one or more `QueueSessions`. Then ask the JMS Provider to create all queues of the system. Queues are essentially named message buffers. The sessions are then applied to create sender and receiver objects with respect to the previously created queues. These objects allow explicit sending and receiving of messages. As receiving messages blocks the calling thread, we are also interested in the interface `MessageListener` that prescribes the implementation of an asynchronous message handler, i.e., of a method `onMessage(Message m)` to evaluate the current message and implement a particular reaction. An implementation of `MessageListener` must be associated with a `QueueReceiver` and then, the message handler is notified by the JMS Provider each time a new message arrives in the particular message queue.

4.2. Implementation of Port-Based Components. In the following we describe how to implement an assembly of port-based components with asynchronous connectors as a JMS application. The implementation of the hierarchical structure of composite components is briefly discussed afterwards. We consider the JMS standard only, thus we do not go into details of the concrete reference implementation Open MQ [**Mic07**].

An assembly comprises component and connector declarations that can be used to create the objects of a corresponding JMS application. In the following we refer to the names used in our metamodel (cf. Fig. 2.1). An Assembly uses a `QueueConnectionFactory` and a `QueueConnection` to create a `QueueSession` for each component declaration in cmps. In this way, we obtain a single-threaded execution context for each component. Using the connector declarations in conns, the names of all queues required with respect to the connected ports of the assembly are available and might be created using the JMS provider as a named `Queue`. Then, for each queue, a sender `QueueSender` and a receiver `QueueReceiver` is created. With the receiver objects we associate component implementations that implement the interface `MessageListener` according to messages of the provided interfaces of all of its ports. In order to enable the sending of messages within these implementations, a reference to all sender objects that relate to the ports, i.e., to the output queues of the particular component is passed to the constructor. After the construction phase is completed, an explicit command `connection.start()` is used to start the message exchange as implemented by the components within the given assembly.

Besides the administrative role of an Assembly, it is the implementation of a component's Behaviour as a `MessageListener` that is important for a realisation using JMS. The implementation may use a state pattern and we propose additionally to pass the sender objects of the component's output queues as a parameter to the constructor of the `MessageListener` implementation. In this way, an initial sending of messages can be implemented in the constructor body. Message construction means to create a corresponding signal object that is passed within an `ObjectMessage`. Of course this is not the only way to pass data within messages in JMS and other formats might be more

suitable for a concrete application. Note that messages are not exchanged unless the underlying connection with the JMS provider was explicitely started. The implementation of the message handler realises the reactions to incoming messages up from the first receive transition within the component's behaviour. Mixed choice states could be realised with timeouts. Finally, an invocation of the message handler with a message not expected in the current state may result in an exception. As `Queues` are by default FIFO buffers, this refers to a system state with a queue whose topmost element was not expected by the registered receiver. In the realm of CFSM systems, such a system state is called an "unspecified reception state".

The following steps summarise the procedure applied by an assembly to set up and start the asynchronous message exchange among its subcomponents.

(1) Get connection to JMS Provider (middleware)
(2) Use connection to create sessions, one for each component
(3) Create queues, one for each port, as administered objects
(4) Use sessions to create sender and receiver for each queue
(5) Create components and register them as asynchronous receiver
(6) Start message exchange (`connection.start()`)

A composite component refers to an assembly in our metamodel; cf. 2.1. We believe that the hierarchical structure can be preserved within an JMS implementation, if composite components implement a `MessageListener` with regard to their relay ports by delegation to the open ports of the assembly.

CHAPTER 3

Behavioural Neutrality in Synchronous Assemblies

1. Syntactical Reduction of Synchronous Assemblies 35
1.1. Behavioural Neutrality and Leaf Components 36
1.2. Reduction Strategy for Synchronous Assemblies 38
2. Port-Based Views of Component Behaviour 39
2.1. Weakly Deterministic IOTSs and Neutrality 39
2.2. Port-Based Computation of Neutral Leaves 41
3. Application to the Compressing Proxy System 43
4. Discussion and Related Work 44
4.1. Neutral Leaf Components and Context Constraints 44
4.2. Neutrality and PADL compatibility 45

It is the goal of this chapter's study to provide support for the construction and behavioural analysis of large scale component-based systems that communicate solely by synchronous communication. Component behaviour can be considered to be *neutral* within a given assembly if its composition does not show any effect on the assembly behaviour apart from the synchronisation of shared transitions without loosing any transition of the original assembly behaviour. A major result of this chapter is an algorithm that shows how to remove neutral components for the computation of a syntactically reduced but behaviourally equivalent assembly. A reduced assembly may then be used to implement a given composite component in a more efficient way. Moreover, we show that it is sufficient to perform neutrality checks using the projection of a component behaviour to one of its ports, provided that the projected behaviour is a weakly deterministic I/O-transition system. The port projection is usually a simpler and smaller transition system, at least after minimisation with respect to greybox respectively blackbox equivalence. Finally, we apply our approach to the compressing proxy system introduced in Chap. 2.

1. Syntactical Reduction of Synchronous Assemblies

Construction and analysis of assembly behaviours usually implies the necessity to compute a product of the underlying transition systems. Then, we often face the state explosion problem that has been the source of investigation at many places. There are, for instance, approaches to compositional reachability analysis (CRA) such as [**CK96**] and, even more prominent under the umbrella of component-based systems, approaches that focus on efficient analysis of safety and liveness properties such as [**BCD02, CFN03, GTBF03, GS05, GGMC$^+$07**]. However, the mentioned studies focus essentially on the analysis of assembly behaviour, whereas our focus is also on the construction of the blackbox behaviour of composite components which encapsulate an assembly. Given a composite component CC that encapsulates an assembly a with behaviour $beh(a)$, the blackbox behaviour $beh(CC)\xi$ of CC is formally defined by observational abstraction (hiding internal transitions) from $beh(CC)$ and thus from $beh(a)$. The idea is then to identify components within the assembly a which do not play any role for the blackbox behaviour of CC. Formally, this idea will be reflected by our notion of *behavioural neutrality*. Then, instead of computing the behaviour $beh(a)$ of the complete assembly it is sufficient to compute the

behaviour of a simpler assembly, say a', where the behaviourally neutral components have been removed from a, such that $beh(a')$ and $beh(a)$ are blackbox equivalent. As a consequence, $beh(CC)$ is blackbox equivalent to the observational abstraction of $beh(a')$. The reduction strategy is particularly useful for acyclic topologies containing a distinguished "rooted" component, but is also sound for arbitrary topologies. The cost of the reduction algorithm depends primary on the cost of the neutrality checks which are performed between connected components of the assembly and hence depends on the complexity of the behaviours of the subcomponents.

We consider assemblies which comprise synchronous connectors only. An assembly a is called *(completely) synchronous*, if $acs(a) = \emptyset$.

1.1. Behavioural Neutrality and Leaf Components. For the formal definition of behavioural neutrality we have to take into account that if we remove components and their connectors from an assembly, components with artificially opened ports remain. These ports should not be available for new connections which is modelled by using unary connectors. Behavioural neutrality is now defined in terms of greybox equivalence between the behaviour of an assembly with two connected components and the behaviour of an assembly with a single component and the binary connector replaced by a unary one. Figure 3.1 illustrates this situation for components $c : C$, $d : D$ and a binary connector $k : (c.p : P, d.q : Q)$, which is replaced by an unary connector $k : (c.p : P)$.

Definition 3.1 *[Neutrality] Let* $a = \langle c : C[p : P], d : D[d : D]; k : K\rangle$ *be a synchronous assembly with* $ports(K) = \{c.p : P, d.q : Q\}$*. The synchronisation via* $k : K$ *in* a *is* behaviourally neutral *for* $c : C$*, if*

$$beh(a) \approx^{\mathrm{gb}} beh(\langle c : C; k : (c.p : P)\rangle) .$$

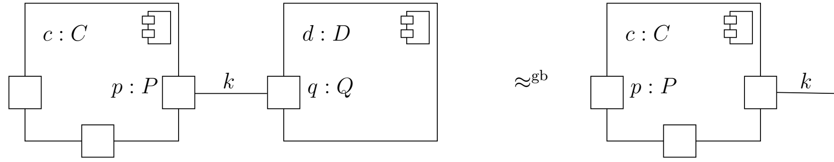


FIGURE 3.1. Definition of neutrality

Component and connector names in assemblies are unique, therefore we may also write slightly shorter "synchronisation via k in a for c" if the context of an assembly is given. Note that behavioural neutrality for $c : C$ can only hold, if the particular communication partner is a single-port component such as $d : D$ in Fig. 3.1. Otherwise the labellings of the compared assemblies would not equal each other which is a necessary prerequisite to apply greybox equivalence for the comparison of the underlying IOTSs. In fact, Def. 3.1 lifts a notion of neutrality between IOTSs to component systems, which is formally defined as follows: An IOTS B is called *neutral* for an IOTS A, if A and B are composable and $A\theta_S \approx^{\mathrm{gb}} A \otimes B$ where $S = L(A) \bowtie L(B)$ are the shared labels of A and B and θ_S internalises S in A. Intuitively, neutrality means that B does not restrict the behaviour given by A if B is composed with A.

Based on the notion of behavioural neutrality, we describe our reduction strategy for assemblies: If the synchronisation at the border of an acyclic assembly topology is behaviourally neutral for the next component attached to it, this leaf can not have behavioural effect on the remaining assembly. We can thus reduce the assembly by removing a neutral leaf component. In order to obtain again a syntactically well-formed assembly, the particular binary connector is removed and replaced by a unary connector (Lem. 3.2). In a second step we eliminate also the unary connector by hiding the port to which the neutral leaf was connected (Lem. 3.3).

Formally, the leaves of an assembly are component declarations which are connected by exactly one binary assembly connector and do not show open ports: For $a \in \mathrm{Asm}$ we let $\mathit{leaves}(a) \subseteq \mathit{cmps}(a)$ such that $d : D \in \mathit{leaves}(a)$, if, and only if, $\mathit{open}(a) \cap \mathit{ports}(d : D) = \emptyset$ and $|\{k : K \mid k : K \in \mathit{conns}(a) \wedge |\mathit{ports}(K)| = 2 \wedge \mathit{ports}(K) \cap \mathit{ports}(d : D) \neq \emptyset\}| = 1$.

Lemma 3.2 (Reduce) *Let $a = \langle c : C[p : P], d : D[q : Q], \mathcal{C}; k : K, \mathcal{K}\rangle$ be a synchronous assembly with $d : D \in \mathit{leaves}(a)$, $\mathcal{C} \subseteq \mathrm{CmpDcl}$, $\mathit{ports}(K) = \{c.p : P, d.q : Q\}$ and $\mathcal{K} \subseteq \mathrm{ConnDcl}$. If the synchronisation via $k : K$ in a is behaviourally neutral for $c : C$, then $\mathit{beh}(a) \approx^{\mathrm{gb}} \mathit{beh}(\langle c : C, \mathcal{C}; k : (c.p : P), \mathcal{K}\rangle)$.*

PROOF. The claim follows from the definitions and the congruence of greybox equivalence w.r.t. relabelling and products. By definition of assembly behaviours, we have that

$$\begin{aligned} \mathit{beh}(a) &= \mathit{beh}(c : C)\sigma \otimes \mathit{beh}(d : D)\sigma \otimes \textstyle\bigotimes_{c' : C' \in \mathcal{C}} \mathit{beh}(c' : C')\sigma\ , \\ \mathit{beh}(\langle c : C, \mathcal{C}; k : (c.p : P), \mathcal{K}\rangle) &= \mathit{beh}(c : C)\sigma' \otimes \textstyle\bigotimes_{c' : C' \in \mathcal{C}} \mathit{beh}(c' : C')\sigma' \end{aligned}$$

with synchronisations $\sigma = \sigma_{(\{c.p, d.q\}, k)} \cup \sigma_{\mathcal{K}}$ and $\sigma' = \sigma_{(c.p, k)} \cup \sigma_{\mathcal{K}}$. Since synchronisation via k in a is neutral for $c : C$ we have $\mathit{beh}(c : C)\sigma_{(c.p,k)} \approx^{\mathrm{gb}} \mathit{beh}(c : C)\sigma_{(\{c.p,d.q\},k)} \otimes \mathit{beh}(d : D)\sigma_{(\{c.p,d.q\},k)}$. Thus, by Lem. 2.16(2), we have $\mathit{beh}(c : C)\sigma' \approx^{\mathrm{gb}} \mathit{beh}(c : C)\sigma \otimes \mathit{beh}(d : D)\sigma$. As neither $\sigma_{(c.p,k)}$ nor $\sigma_{(\{c.p,d.q\},k)}$ change $\bigotimes_{c' : C' \in \mathcal{C}} \mathit{beh}(c' : C')$, we also have $\bigotimes_{c' : C' \in \mathcal{C}} \mathit{beh}(c' : C')\sigma = \bigotimes_{c' : C' \in \mathcal{C}} \mathit{beh}(c' : C')\sigma'$, and the claim follows by applying Lem. 2.16(3). □

Let $C \in \mathrm{Cmp}$ and $p : P \in \mathit{ports}(C)$. The *port hiding* of $p : P$ in C, written $C \setminus (p : P)$, results in a component $C' \in \mathrm{Cmp}$ such that $\mathit{ports}(C') = \mathit{ports}(C) \setminus \{p : P\}$ and $\mathit{beh}(C') = \mathit{beh}(C)/\{p.m \mid m \in \mathit{msg}(P)\}$.

Lemma 3.3 (Hide) *Let $a = \langle c : C[p : P], \mathcal{C}; k : (c.p : P), \mathcal{K}\rangle$ be a synchronous assembly. Then $\mathit{beh}(a)/\{k.m \mid m \in \mathit{msg}(P)\} \approx^{\mathrm{gb}} \mathit{beh}(a[c : C \mapsto C'])$, where $C' = C \setminus (p : P)$.*

PROOF. The claim follows from the definitions using the fact that the labels $\{k.m \mid m \in \mathit{msg}(P)\}$ are not shared with any component of the reduced assembly, and that greybox equivalence is a congruence w.r.t. IOTS products. By definition of assembly behaviours we have for the left-hand side

$$\mathit{beh}(a)/H = (c.\mathit{beh}(C)\sigma \otimes \textstyle\bigotimes_{c' : C' \in \mathcal{C}} \mathit{beh}(c' : C')\sigma)/H,$$

and by definition of type substitution in assemblies and hiding of ports for the right-hand side

$$\begin{aligned} &\mathit{beh}(a[c : C \mapsto (C \setminus (p : P))]) \\ &\quad = c.\mathit{beh}(C \setminus (p : P))\sigma_{\mathcal{K}} \otimes \textstyle\bigotimes_{c' : C' \in \mathcal{C}} \mathit{beh}(c' : C')\sigma_{\mathcal{K}} \\ &\quad = c.(\mathit{beh}(C)/\{p.m \mid m \in \mathit{msg}(P)\})\sigma_{\mathcal{K}} \otimes \textstyle\bigotimes_{c' : C' \in \mathcal{C}} \mathit{beh}(c' : C')\sigma_{\mathcal{K}}, \end{aligned}$$

with synchronisations $\sigma = \sigma_{(c.p,k)} \cup \sigma_{\mathcal{K}}$ and hiding $H = \{k.m \mid m \in \mathit{msg}(P)\}$. As H does not affect $\mathcal{C}$ and $\mathcal{K}$, the right-hand side of the first equation is greybox equivalent to $c.\mathit{beh}(C)\sigma/H \otimes \bigotimes_{c' : C' \in \mathcal{C}} \mathit{beh}(c' : C')\sigma$ by Lem. 2.17(3). Now $c.\mathit{beh}(C)\sigma/H = (c.\mathit{beh}(C)\sigma_{(c.p,k)}/H)\sigma_{\mathcal{K}}$. By definition of unary synchronisation relabellings we have for $L_{\mathit{beh}(C)} = (I, O, T)$ and $L_{\mathit{beh}(C)\sigma_{(c.p,k)}} = (I', O', T')$ that $k.m \in T'$ iff $p.m \in I \cup O$. Thus we obtain

$$c.\mathit{beh}(C)\sigma_{(c.p,k)}/H \approx^{\mathrm{gb}} c.(\mathit{beh}(C)/\textstyle\bigcup L_{\mathit{beh}_{p:P}(C)}).$$

Since removing $\sigma_{(c.p,k)}$ from σ does not change $\bigotimes_{c' : C' \in \mathcal{C}} \mathit{beh}(c' : C')$, we have $\bigotimes_{c' : C' \in \mathcal{C}} \mathit{beh}(c' : C')\sigma = \bigotimes_{c' : C' \in \mathcal{C}} \mathit{beh}(c' : C')\sigma_{\mathcal{K}}$ and the claim follows by Lem. 2.16(3). □

1.2. Reduction Strategy for Synchronous Assemblies. The lemma for reduction and hiding allow to remove neutral leaf components one after the other. The *neutral leaves* of an assembly a are defined by $\mathit{neutralleaves}(a) \subseteq \mathit{cmps}(a)$ such that $d : D \in \mathit{neutralleaves}(a)$, if, and only if, there are component declarations $c : C[p : P], d : D[q : Q] \in \mathit{cmps}(a)$, $d : D \in \mathit{leaves}(a)$, $k : (c.p : P, d.q : Q) \in \mathit{conns}(a)$ and the synchronisation in a via $k : (c.p : P, d.q : Q)$ is neutral for $c : C$. An assembly a is called *finite* if $\mathit{cmps}(a)$ is finite. Now define the *syntactical reduction* of a finite assembly $a \in \mathrm{Asm}$ by $\mathit{red} : \mathrm{Asm} \to \mathrm{Asm}$ such that $\mathit{red}(a)$ is the assembly computed by the following algorithm:

```
while neutralleaves(a) ≠ ∅
do d : D ← choose from neutralleaves(a)
    let a = ⟨c : C, d : D, 𝒞; k : (c.p : P, d.q : Q), 𝒦⟩ an assembly
            such that synchronisation via k in a is neutral for c
    a ← a[c : C ↦ (C \ (p : P))]
od
```

Of course, in general, the assembly a and the reduced assembly $\mathit{red}(a)$ will not be greybox equivalent because by hiding of ports $\mathit{red}(a)$ shows less communication than a. The assemblies, however, have observationally equivalent behaviours.

Lemma 3.4 (Assembly) *Let a be a finite assembly. Then $\mathit{beh}(a) \approx^{\mathrm{bb}} \mathit{beh}(\mathit{red}(a))$.*

PROOF. The algorithm terminates for each finite assembly a. For each iteration of the `while`-loop let a_0 be the assembly on entry and a_1 the assembly on exit. We show $\mathit{beh}(a_0) \approx^{\mathrm{bb}} \mathit{beh}(a_1)$. Let $a_0 = \langle c : C, d : D, \mathcal{C}; k : (c.p : P, d.q : Q), \mathcal{K}\rangle$ such that the synchronisation via k in a is neutral for c, then $a_1 = a[c : C \mapsto (C \setminus (p : P))]$. Assume $H = \{k.m \mid m \in \mathit{msg}(P)\}$, then the following holds:

$$\begin{aligned} \mathit{beh}(a_0)/H &\approx^{\mathrm{gb}} \mathit{beh}(\langle c : C, \mathcal{C}; k : (c.p : P), \mathcal{K}\rangle)/H && \text{[Lem. 3.2, Lem. 2.16(1)]} \\ &\approx^{\mathrm{gb}} \mathit{beh}(a[c : C \mapsto (C \setminus (p : P))]) = \mathit{beh}(a_1)\ . && \text{[Lem. 3.3]} \end{aligned}$$

Thus, $\mathit{beh}(a_0)/H \approx^{\mathrm{gb}} \mathit{beh}(a_1)$ and since H contains internal labels only we may also hide all internal labels, that is $\mathit{beh}(a_0)\xi \approx^{\mathrm{gb}} \mathit{beh}(a_1)\xi$ and hence we obtain the invariant $\mathit{beh}(a_0) \approx^{\mathrm{bb}} \mathit{beh}(a_1)$. The claim follows from the invariant. □

The reduction strategy for assemblies is directly applicable to the computation of a blackbox composite component behaviour.

Theorem 3.5 (Reduced composite component) *Let $\langle a; \mathcal{P}; \mathcal{K}\rangle \in \mathrm{CCmp}$ with a finite. Then $\mathit{beh}(\langle a; \mathcal{P}; \mathcal{K}\rangle) \approx^{\mathrm{bb}} \mathit{beh}(\langle \mathit{red}(a); \mathcal{P}; \mathcal{K}\rangle)$.*

PROOF. By definition, $\mathit{leaves}(a)$ do not have open ports. Thus no components with a delegate connector to a relay port can be removed from a by the reduction algorithm. Therefore, $\langle \mathit{red}(a); \mathcal{P}; \mathcal{K}\rangle$ is a well-formed composite component. Blackbox equivalence does not distinguish between internal and τ transitions, thus $\approx^{\mathrm{gb}}$ and $\approx^{\mathrm{bb}}$ coincide and the result follows from Lem. 3.4 by renaming actions on open ports of a, and hence of $\mathit{red}(a)$, to actions on the relay ports $\mathcal{P}$. □

Our reduction strategy is sound for arbitrarily structured finite assemblies. But it is particularly useful for assemblies with an *acyclic* topology and for composite components which are *rooted*, i.e., exactly one of the assembly components is connected to the relay ports of the composite component, like in Fig. 3.2. Then, in the best case, all subcomponents but the root $\mathsf{rc} : \mathsf{RC}$ can be removed such that $\mathit{beh}(\mathsf{CC}) \approx^{\mathrm{gb}} \mathit{beh}(\mathsf{rc}{:}\mathsf{RC})\rho$ where ρ is a relay relabelling. Note that acyclic topologies do not present a too severe restriction in terms of practical applicability. There is, e.g., a direct match with the architectural pattern “Blackboard” [**BFH⁺07**] for the design of distributed communications using a common data structure. Also, the architectural design of a large part of the Common Component Modelling Example [**RRMP08**] adheres to an even more restricted acyclic structure, called

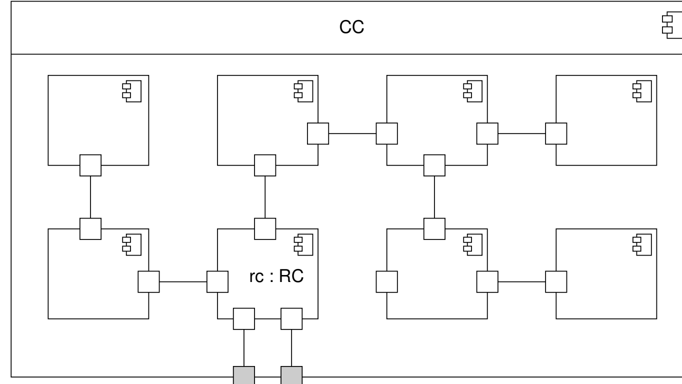


FIGURE 3.2. Composite component with acyclic assembly

"star topology" in [**BCD02**], where each leaf is attached to one centre component. Our reduction strategy can also be effectively applied to assembly topologies containing cycles as long as there are neutral leaves around that can be removed. When a cycle is reached we would propose to encapsulate the cycle into a new, nested composite component and to proceed as before by searching again for neutral leaves.

Finally, we would like to stress again that the behaviour of a reduced assembly is not greybox equivalent to the behaviour of the original assembly and that the results of our approach rely fundamentally on the additional abstraction step that is applied when the blackbox behaviour of a composite component is considered. Not being greybox equivalent means for the case of assemblies that the systems may not show the same synchronisations. Indeed, syntactical reduction of assemblies removes neutral components and in the resulting assembly the synchronisation is merely mimicked by internalisation of the particular transitions to which the removed components where neutral. Now it is important to note that our reduction uses the IOTS product operator to decide on neutrality and one of the characteristic properties of IOTS composition is that only successful communication is preserved. In case of shared transitions that are not synchronised the particular transition is not considered any more. In particular this means, that the removal of neutral components may also remove behaviour which might still be of interest within a global analysis of the given assembly. Therefore, to ensure the correctness of an analysis of a reduced assembly one needs to check that the property under investigation does not rely on unsuccessful communications.

2. Port-Based Views of Component Behaviour

A closer look reveals that in order to decide on neutrality along Def. 3.1, one needs to compute the composition of a leaf component behaviour with the behaviour of the attached component, which can still be quite expensive. To further optimise our method we are interested in criteria to avoid composition of full behaviours of components and to compose smaller transition systems instead. An immediate candidate for smaller transition systems are port-based views of the component behaviour. In Sect. 2.2 we show that port-based views can be used to provide such criteria, if they are weakly deterministic. For this purpose we first proceed with some definitions and analysis on the level of I/O-transition systems. Afterwards, the results are related back to the component level.

2.1. Weakly Deterministic IOTSs and Neutrality. Port-based views of component behaviour hide transitions which do not involve messages of the particular port. Aiming at a decision on neutrality between component behaviours using port-based views then means to check that such a hiding does not hide too much. Intuitively we need to make sure that

the behaviour at hidden ports does not involve decisions which compromises the correctness of a neutrality analysis at a different port. Since hiding means a relabelling to τ and neutrality for IOTSs is based on greybox equivalence such a critical decision is an "internal choice" which can not be removed without invalidating greybox equivalence. Hence, on the one hand we want to allow for τ-transitions but on the other hand we do not allow for non-removable internal choice transitions. It turns out that Milner's (weak) determinacy [**Mil89**, Def. 11.3, Ex. 11.1] yields an appropriate notion, which is transfered to IOTSs by a definition of "weak determinism". Weak determinacy is preserved by observational equivalence [**Mil89**, Prop. 11.4] which helps to provide a quite intuitive understanding of weak determinism for IOTSs. In [**HJK08**] we prove that an IOTS A is weakly deterministic, if there is a greybox equivalent, minimal IOTS B without τ-transitions; such a B is a weakly deterministic IOTS without τ-transitions and hence also strongly deterministic. By the transition minimisation procedure of Eloranta [**Elo91**], a finite τ-free minimal IOTS is unique up to graph-isomorphism. Thus, for ensuring that a finite IOTS is weakly deterministic, it suffices to minimise the IOTS w.r.t. the number of states and transitions and to check that the resulting IOTS has no τ-transitions and that from each state there is at most one transition for each label.

For the precise definition of weakly deterministic IOTSs we need to consider the weak traces of an IOTS. Given an IOTS $A = (L, S, s_0, \Delta)$, if we remove in each finite trace of A all τ occurrences then we obtain the weak traces of A. We define the traces of an IOTS based on its initial execution fragments. Let A be an IOTS. The trace of an execution fragment $\rho \in \mathit{frag}_0^*(A)$ is the sequence of labels occurring in ρ, i.e. the trace of $\rho = (s_0 l_1 s_1 \ldots l_n s_n)$ is the sequence $\lambda = l_1 \ldots l_n \in \mathcal{L}(A)^*$. A weak trace is obtained from λ by projection to the alphabet $\bigcup L_A$, i.e. by removing all τs from $l_1 \ldots l_n$.[1] Note that the empty sequence ϵ is also a weak trace. With the weak traces for all initial execution fragments $\rho \in \mathit{frag}_0^*(A)$ we obtain the set of weak traces of A, denoted by $\mathscr{T}^*(A)$.

Lemma 3.6 *If A and B are IOTSs with $A \approx^{\mathrm{gb}} B$, then $\mathscr{T}^*(A) = \mathscr{T}^*(B)$.*

PROOF. Greybox equivalent IOTSs have, up to occurrences of τ transitions, the same finite traces. Hence they have the same weak traces. □

An IOTS A is weakly deterministic if all finite traces of A which coincide up to τ transitions lead to greybox equivalent elements of A. We extend the notation $(s, l, s') \in \Delta_A^{\mathrm{X}}$ to label sequences, such that $(s, \lambda, s') \in \Delta_A^{\mathrm{X}}$ if s' is reachable via labels in λ (in the given order), possibly interspersed with labels in X. Then $(s, \epsilon, s') \in \Delta_A^{\mathrm{X}}$ is equivalent with $(s, s') \in \Delta_A^{\mathrm{X}}$.

Definition 3.7 (Weakly deterministic) *An IOTS $A = (L, S, s_0, \Delta)$ is* weakly deterministic *if for all weak traces $\lambda \in \mathscr{T}^*(A)$ the following holds: If $(s_0, \lambda, s_1) \in \Delta^\tau$ and $(s_0, \lambda, s_2) \in \Delta^\tau$ then $s_1 \approx^{\mathrm{gb}} s_2$.*

Remember that an IOTS B is called *neutral* for an IOTS A, if A and B are composable and $A\theta_S \approx^{\mathrm{gb}} A \otimes B$ where $S = L(A) \bowtie L(B)$ are the shared labels of A and B and θ_S internalises S in A. We extend a sufficient criterion, called "interface theorem", from Cheung and Kramer [**CK96**] for determining neutrality to our setting. We generalise their assumptions to include not necessarily finite IOTS and use greybox equivalence and weakly deterministic IOTSs.

Proposition 3.8 *Let A and B be two composable IOTSs with $\bigcup L_B \subseteq \bigcup L_A$, and let $H = \bigcup L_A \setminus \bigcup L_B$. Let B be weakly-deterministic. Then B is neutral for A if, and only if, $\mathscr{T}^*(A/H) \subseteq \mathscr{T}^*(B)$.*

PROOF. We abbreviate $\theta_{\mathcal{S}(A,B)}$ by $\theta_{\mathcal{S}}$. "⇐" We show that $R = \{(s_A, (s_A, s_B)) \in S_A \times (S_A \times S_B) \mid \exists \lambda \in \bigcup {L_A}^* \,.\, (s_{0,A}, \lambda, s_A) \in \Delta_A^\tau \wedge (s_{0,B}, \lambda/H, s_B) \in \Delta_B^\tau\}$, where

[1] Remember that $\mathcal{L}(A) = I_A \cup O_A \cup T_A \cup \{\tau\}$ and $\bigcup L_A = I_A \cup O_A \cup T_A$

λ/H is obtained from λ by removal of all labels given by H, is a witness for greybox equivalence between $A\theta_S$ and $A \otimes B$.

Let $\mathscr{T}^*(A/H) \subseteq \mathscr{T}^*(B)$ hold. Let $A = (L_A, S_A, s_{0,A}, \Delta_A)$, $B = (L_B, S_B, s_{0,B}, \Delta_B)$, $A\theta_{\mathcal{S}} = (L_A\theta_{\mathcal{S}}, S_A, s_{0,A}, \Delta_{A\theta_{\mathcal{S}}})$, $A \otimes B = (L_A \otimes L_B, S_A \times S_B, (s_{0,A}, s_{0,B}), \Delta_{A\otimes B})$. Let $R = \{(s_A, (s_A, s_B)) \in S_A \times (S_A \times S_B) \mid \exists \lambda \in \bigcup L_A{}^* \,.\, (s_{0,A}, \lambda, s_A) \in \Delta_A^\tau \wedge (s_{0,B}, \lambda/H, s_B) \in \Delta_B^\tau\}$ where λ/H is the sequence of labels which results from λ when removing all labels in H. Then $(s_{0,A}, (s_{0,A}, s_{0,B})) \in R$. In order to show that R is a witness for greybox equivalence between $A\theta_{\mathcal{S}}$ and $A \otimes B$, let $(s_A, (s_A, s_B)) \in R$.

Let $(s_A, l, s'_A) \in \Delta_{A\theta_{\mathcal{S}}}$. If $l \in H$ or $l = \tau$, then $((s_A, s_B), l, (s'_A, s_B)) \in \Delta_{A\otimes B}$ and $(s'_A, (s'_A, s_B)) \in R$. If $l \in \mathcal{S}(A, B)$, let $\lambda \in \bigcup L_A{}^*$ be a weak trace with $(s_{0,A}, \lambda, s_A) \in \Delta_A^\tau$ and $(s_{0,B}, \lambda/H, s_B) \in \Delta_B^\tau$. Then $\lambda l \in \mathscr{T}^*(A)$ and thus $(\lambda/H)l = (\lambda l)/H \in \mathscr{T}^*(A/H) \subseteq \mathscr{T}^*(B)$. Hence there is a $s_B^{(1)} \in S_B$ such that $(s_{0,B}, \lambda/H, s_B^{(1)}) \in \Delta_B^\tau$ and $(s_B^{(1)}, l, s_B^{(2)}) \in \Delta_B$. As B is weakly deterministic, $s_B^{(1)} \approx^{\mathrm{gb}} s_B$, and thus there is a $s'_B \in S_B$ with $(s_B, l, s'_B) \in \Delta_B^\tau$. Thus $((s_A, s_B), l, (s'_A, s'_B)) \in \Delta_{A\otimes B}^\tau$ and $(s'_A, (s'_A, s'_B)) \in R$.

Let $((s_A, s_B), l, (s'_A, s'_B)) \in \Delta_{A\otimes B}$. Then either $l = \tau$ or $l \in \mathcal{S}(A, B)$ or $l \in H$, since $\bigcup L_B \subseteq \bigcup L_A$. If $(s_A, \tau, s'_A) \in \Delta_A$ then $s'_B = s_B$, $(s_A, \tau, s'_A) \in \Delta_{A\theta_{\mathcal{S}}}$ and $(s'_A, (s'_A, s_B)) \in R$; if $(s_B, \tau, s'_B) \in \Delta_B$ then $s'_A = s_A$, $(s_A, s_A) \in \Delta_{A\theta_{\mathcal{S}}}^\tau$, and $(s_A, (s_A, s'_B)) \in R$. If $l \in \mathcal{S}(A, B)$, then $(s_A, l, s'_A) \in \Delta_{A\theta_{\mathcal{S}}}$ and $(s'_A, (s'_A, s'_B)) \in R$. If $l \in H$, then $(s_A, l, s'_A) \in \Delta_{A\theta_{\mathcal{S}}}$, $s'_B = s_B$, and $(s'_A, (s'_A, s_B)) \in R$.

"$\Rightarrow$" The claim follows from Lem. 3.6 using the fact that H does not contain shared labels of A and B and the congruence of greybox equivalence w.r.t. hiding. Let $A\theta_{\mathcal{S}} \approx^{\mathrm{gb}} A \otimes B$. Then $A/H \otimes B \approx^{\mathrm{gb}} (A \otimes B)/H$ by Lem. 2.17 (3), as $\mathcal{S}(A, B) \cap H = \emptyset$. Furthermore $(A\theta_{\mathcal{S}})/H \approx^{\mathrm{gb}} (A/H)\theta_{\mathcal{S}}$, again since $\mathcal{S}(A, B) \cap H = \emptyset$. Thus $(A/H)\theta_{\mathcal{S}} \approx^{\mathrm{gb}} A/H \otimes B$ by Lem. 2.16 (1). Since $\bigcup L_{A/H} = \bigcup L_B$, all labels between A/H and B are shared, and thus $\mathscr{T}^*(A/H \otimes B) \subseteq \mathscr{T}^*(B)$. By Lem. 3.6, we have $\mathscr{T}^*(A/H) = \mathscr{T}^*((A/H)\theta_{\mathcal{S}}) = \mathscr{T}^*(A/H \otimes B) \subseteq \mathscr{T}^*(B)$. □

The following corollary is the crucial result needed for the port-based computation of neutral leaf components discussed hereafter. Essentially it says that neutrality of a weakly deterministic IOTS B for some IOTS A can be propagated to the neutrality of B for some more complex IOTS C, if A is greybox equivalent to some view C/H on the larger behaviour C.

Corollary 3.9 *Let A, B, and C be IOTSs such that $\bigcup L_B = \bigcup L_A \subseteq \bigcup L_C$. Let B be composable with A and C. Let $H = \bigcup L_C \setminus \bigcup L_A$ and $C/H \approx^{\mathrm{gb}} A$. Let B be weakly deterministic. Then B is neutral for A if, and only if, B is neutral for C.*

PROOF. First, since $C/H \approx^{\mathrm{gb}} A$ we have, by Lem. 3.6, $\mathscr{T}^*(C/H) = \mathscr{T}^*(A)$. B is neutral for A iff (by Prop. 3.8, since $\bigcup L_B = \bigcup L_A$) $\mathscr{T}^*(A) \subseteq \mathscr{T}^*(B)$ iff $\mathscr{T}^*(C/H) \subseteq \mathscr{T}^*(B)$ iff (by Prop. 3.8, taking C for A) B is neutral for C. □

2.2. Port-Based Computation of Neutral Leaves. In this section, we focus on the behavioural neutrality checks which are performed component-wise in our reduction algorithm of Sect. 1. Components communicate exclusively via ports; therefore it should be sufficient to compare instead of the complete behaviour of two components only the behaviour visible at their connected ports which are usually much smaller IOTSs. We show that the neutrality checks indeed may be optimised by considering port-based views which hide component actions that are not visible at the given port. Here we would like to stress that port-based checks are an optimisation to compare behaviours; the views are not intended to replace component behaviours in the computation of assembly behaviours. Of course, such an approach can only be sound if the particular view preserves enough information of the complete component behaviour such that neutrality between port-based views of component behaviours indeed implies neutrality for the synchronisation of the

underlying complete behaviours. It will turn out that weakly determinism is a fundamental prerequisite.

Definition 3.10 (Port behaviour) *Let $C \in \mathrm{Cmp}$ with $p : P \in ports(c : C)$. The* behaviour *of C at port $p : P$ is given by $beh(C)_{p:P} = beh(C)/(\bigcup(L_C) \setminus \{p.m \mid m \in msg(P)\})$. The behaviour $beh(C)_{p:P}$ is also called* port behaviour *of $p : P$ in the context of C.*

Similar to the hiding of ports, we may construct new components using the behaviour of a component at a particular port. Let $C \in \mathrm{Cmp}$ and $p : P \in ports(C)$. The *restriction* of C to port $p : P$, written $C{\downarrow_{p:P}}$, yields a component (type) $C' \in \mathrm{Cmp}$, given by $ports(C') = \{p : P\}$ and $beh(C') = beh(C)_{p:P}$.

Definition 3.11 (Port neutrality) *Let $a = \langle c : C[p : P], d : D[d : D]; k : K\rangle$ be a synchronous assembly with $ports(K) = \{c.p : P, d.q : Q\}$. The port behaviour $beh(D)_{q:Q}$ is k-*neutral *for $beh(C)_{p:P}$ if the synchronisation via $k : K$ in $a[c : C \mapsto (C{\downarrow_{p:P}}), d : D \mapsto (D{\downarrow_{q:Q}})]$ is behaviourally neutral for $c : C{\downarrow_{p:P}}$.*

Unfolding the definition of port neutrality results in a characterisation in terms of IOTSs: $beh(C)_{p:P}\sigma \otimes beh(D)_{q:Q}\sigma \approx^{\mathrm{gb}} beh(C)_{p:P}\theta_S$ where σ is an appropriate synchronous relabelling using the connector $k : K$ and $S = \mathcal{S}(beh(C)_{p:P}\sigma, beh(D)_{q:Q}\sigma)$. Hence, port behaviours can only be behaviourally neutral if their labellings are inverse, i.e., if input and output labels mutually coincide. Port neutrality has the important consequence that the behaviour of a component is unaffected if a port of the component is connected to a neutral port of another component, provided that the behaviour of this port is weakly deterministic. Weak determinism is the crucial property required to preserve neutrality within our abstraction from component behaviours to port behaviours as given by Def. 3.10.

The following theorem is an application of the results discussed on the level of I/O-transition systems in Sect. 2.1. It is at the core of a more efficient check for neutrality in assemblies. Note that component C may have more than one port.

Theorem 3.12 (Port-based check) *Let $a = \langle c : C, d : D; k : K\rangle$ be a synchronous assembly, $p : P \in ports(C)$, $ports(D) = \{q : Q\}$ and $ports(K) = \{c.p : P, d.q : Q\}$. If $beh(D)_{q:Q}$ is weakly deterministic and $beh(D)_{q:Q}$ is k-neutral for $beh(C)_{p:P}$, then the synchronisation via $k{:}K$ in a is behaviourally neutral for $c : C$.*

PROOF. By expanding definitions we show that the theorem is an instance of Cor. 3.9. By definition of port neutrality Def. 3.11 we need to show that the behavioural neutrality of the synchronisation via $k : K$ in $a[c : C \mapsto (C{\downarrow_{p:P}}), d : D \mapsto (D{\downarrow_{q:Q}})]$ implies the behavioural neutrality of the synchronisation via k:K in a for $c : C$. By definition of neutrality Def. 3.1 this amounts to show that

$$beh(\langle c : C{\downarrow_{p:P}}, d : D{\downarrow_{q:Q}}; k : (c.p : P, d.q : Q)\rangle) \approx^{\mathrm{gb}} beh(\langle c : C{\downarrow_{p:P}}; k : (c.p : P)\rangle)$$
$$\text{implies} \quad beh(\langle c : C, d : D; k : (c.p : P, d.q : Q)\rangle) \approx^{\mathrm{gb}} beh(\langle c : C; k : (c.p : P)\rangle).$$

By definition of assembly behaviours:

$$beh(c : C{\downarrow_{p:P}})\sigma_2 \otimes beh(d : D{\downarrow_{q:Q}})\sigma_2 \approx^{\mathrm{gb}} beh(c : C{\downarrow_{p:P}})\sigma_1$$
$$\text{implies} \quad beh(c : C)\sigma_2 \otimes beh(d : D)\sigma_2 \approx^{\mathrm{gb}} beh(c : C)\sigma_1$$

with $\sigma_2 = \sigma_{(c.p,d.q,k)}$ and $\sigma_1 = \sigma_{(c.p,k)}$. By definition of restriction and behaviours of component declarations we obtain

$$(c.beh(C)_{p:P})\sigma_2 \otimes (d.beh(D)_{q:Q})\sigma_2 \approx^{\mathrm{gb}} (c.beh(C)_{p:P})\sigma_1$$
$$\text{implies} \quad (c.beh(C))\sigma_2 \otimes (d.beh(D))\sigma_2 \approx^{\mathrm{gb}} (c.beh(C))\sigma_1$$

which is, by $ports(D) = \{q : Q\}$, equivalent to

$$(c.beh(C)_{p:P})\sigma_2 \otimes (d.beh(D))\sigma_2 \approx^{\mathrm{gb}} (c.beh(C)_{p:P})\sigma_1$$
$$\text{implies} \quad (c.beh(C))\sigma_2 \otimes (d.beh(D))\sigma_2 \approx^{\mathrm{gb}} (c.beh(C))\sigma_1$$

By definition of port behaviours we have $beh(C)/H = beh(C)_{p:P}$ with $H = \bigcup L_C \setminus \{p.m \mid m \in msg(P)\}$, i.e., we have to prove that

$$(c.beh(C)/H)\sigma_2 \otimes (d.beh(D))\sigma_2 \approx^{\mathrm{gb}} (c.beh(C)/H)\sigma_1$$
$$\text{implies} \quad (c.beh(C))\sigma_2 \otimes (d.beh(D))\sigma_2 \approx^{\mathrm{gb}} (c.beh(C))\sigma_1$$

Now H does not interfere with σ_1 and σ_2. Therefore, by $A \equiv (c.beh(C))\sigma_2/H$, $B \equiv (d.beh(D))\sigma_2$ and $C \equiv (c.beh(C))\sigma_1$ we have an instance of Cor. 3.9. □

The theorem allows for a port-based computation of the neutral leaves of an assembly which links our analysis back to the results of Sect. 1.

Corollary 3.13 *Let a be a synchronous assembly, $d : D \in leaves(a)$, $ports(D) = \{q : Q\}$ and $k : (c.p : P, d.q : Q) \in conns(a)$. If $beh(D)_{q:Q}$ is weakly deterministic and $beh(D)_{q:Q}$ is k-neutral for $beh(C)_{p:P}$ then $d : D \in neutralleaves(a)$, where $C = ty(cmp(c.p : P))$.*

PROOF. By Thm. 3.12 we derive neutrality of the leaf component D from the given port neutrality, and with the definition of neutral leaves the claim follows. □

3. Application to the Compressing Proxy System

We apply the results of this chapter, in particular the reduction algorithm of Sect. 1, to the efficient computation of the behaviour of the CompressingProxy component (cf. Fig. 2.2). We start with the underlying assembly which contains the three components adapt:Adaptor, gzip:GZip, and gifToJpg:GifToJpg and assume synchronous connectors.

Let $a = \langle \mathcal{C}; \mathcal{K} \rangle$ be a synchronous assembly, $\mathcal{C} = \{$adapt:Adaptor, gzip:GZip, gif:GifToJpg$\}$ and $\mathcal{K} = \{$tz:(t:TxtCompr, z:Zip), gj: (g: GifCompr, j:ToJpg)$\}$. First, we choose the leaf gzip:GZip and consider its port z:Zip. Since GZip is a single-port component, component behaviour and port behaviour of z:Zip in the context of gzip:GZip coincide. The port behaviour is obviously weakly deterministic, since there are no τ-transitions and the transition system is already (strongly) deterministic; cf. Fig. 2.3. Then we check that $beh(\mathsf{GZip})_{\mathsf{z:Zip}}$ is tz-neutral for $beh(\mathsf{Adaptor})_{\mathsf{t:TxtCompr}}$ by computation of the product

$$beh(\mathsf{Adaptor})_{\mathsf{t:TxtCompr}}\sigma \otimes beh(\mathsf{GZip})_{\mathsf{z:Zip}}\sigma$$

with synchronous relabelling σ according to the connector tz. This product represents the synchronisation via tz in an assembly according to Def. 3.11. By computation of the port behaviour $beh(\mathsf{Adaptor})_{\mathsf{t:TxtCompr}}\sigma$ all transitions of the adaptor behaviour as given in Fig. 2.3 which do not involve port t:TxtCompr are relabelled to τ. The resulting IOTS can be replaced by an observationally equivalent minimised IOTS. Then the product with $beh(\mathsf{GZip})_{\mathsf{z:Zip}}\sigma$ yields an IOTS as given in Fig. 3.3. The product of the port behaviours is greybox equivalent to $(beh(\mathsf{Adaptor})_{\mathsf{t:TxtCompr}}\sigma)\theta_S$ (cf. Fig. 2.3), where the shared labels $S = \{$tz.txt, tz.endTxt, tz.zip, tz.endZip, tz.stop$\}$ are internalised. Hence, by Cor. 3.13, gzip:GZip is a neutral leaf of the assembly a. Our algorithm proceeds by removing this leaf and hiding the port t:TxtCompr, i.e., the component type Adaptor is replaced by the component type Adaptor' obtained from the port hiding Adaptor $\setminus$ (t:TxtCompr). The resulting behaviour of Adaptor' has a large number of τ-transitions and can be minimised to an IOTS with 4 states and 5 transitions. Let a' be the new assembly obtained from a after the first reduction step.

In the second step, we choose the leaf gifToJpg:GifToJpg of a' and consider its port j:ToJpg. Again it is a single-port component but unfortunately, the port behaviour at j:ToJpg is not weakly deterministic, since there is a weak trace gif which leads to state j_1 and also to state j_2 (cf. Fig. 2.3), but these states are not greybox equivalent since in j_1 it is possible to choose cancel and reach state j_0 which does not show an output transition labelled j.jpg. Such an output, however would be required to be in relation with j_2. Thus we cannot apply Cor. 3.13 and we have to show directly on the level of component behaviours that the synchronisation via gj in a' is behaviourally neutral for adapt : Adaptor'. The neutrality

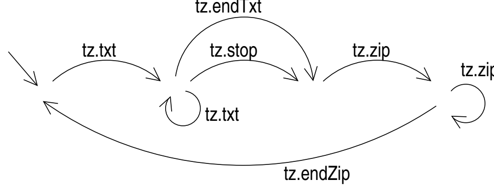


FIGURE 3.3. $beh(\mathsf{Adaptor})_{\mathsf{t:TxtCompr}}\sigma \otimes beh(\mathsf{GZip})_{\mathsf{z:Zip}}\sigma$

check is indeed successful. Our algorithm now removes also the leaf gifToJpg:GifToJpg from a' and hides the port g:GifCompr of Adaptor'. Thus, we obtain as the final result of the reduction an assembly a'' consisting of a single component adaptor : Adaptor" where Adaptor" has only two ports u:UpStream and d:DownStream. The behaviour of Adaptor" has only two states and three transitions after minimisation and hence the same holds for the behaviour of a''.

Finally, we apply Thm. 3.5 which shows that $beh(\mathsf{CompressingProxy})$ is observational equivalent to the behaviour of a composite component which encapsulates the very simple assembly a'' and which results in the behaviour shown in Fig. 2.4b. The most expensive task in our reduction was the neutrality check between gifToJpg:GifToJpg and adapt:Adaptor' which led to the construction of a product IOTS with 12 states and 23 transitions (before minimisation). On the other hand, if we would have directly computed the behaviour of CompressingProxy based on the behaviour of the original assembly a, then the most expensive task would have been the construction of the complete behaviour of a which has 30 states and 65 transitions (also before minimisation).

4. Discussion and Related Work

The neutrality check during the syntactical reduction of an assembly involves the computation of a product at the border of the component assembly for each leaf component. Therefore, compared to the computation of the complete product, there is no direct improvement with respect to the number of considered transition systems. However, besides the construction of syntactically reduced and observationally equivalent component types, the approach can be used as a starting point for analysis of verification problems that do not depend on the local behaviours of removed neutral leaf components.

Our approach and, in particular, our notion of neutral component behaviour shows similarities to work of Cheung and Kramer [**CK96**] and Bernardo et al. [**BCD02**].

4.1. Neutral Leaf Components and Context Constraints. Cheung and Kramer use automatically derived context constraints to construct the behaviour, formalised by labelled transition systems (LTS), of composed systems more efficiently [**CK96**]. Context constraints take the form of interface processes that capture the interplay between a set of composed processes playing the role of an environment for a single fixed process as part of the composition. If the composition of interface and fixed process results in a smaller transition system, it is substituted. The correctness of the approach relies on a transparency property that requires a strong semantic equivalence between a (possibly) composed process P and its composition $P \parallel I$ with the interface process I. Criteria guaranteeing the transparency are identified in an interface theorem [**CK96**, Thm. 6.4.1] to which Prop. 3.8 can be considered the counterpart using IOTSs together with weaker equivalence notions. The criteria of [**CK96**] have been formulated for communicating finite-state processes where parallel composition involves an error state for failing communications and relies on strong traces, strong process equivalence, and strongly deterministic finite-state processes. Even though the possibility of using weaker equivalences is mentioned in [**CK96**, p. 354], it is not elaborated. We have generalised the assumptions to include not necessarily finite IOTSs and

use weak notions of equivalence, traces, and determinism, i.e., greybox equivalence, weak traces, and weakly deterministic IOTSs.

Furthermore, our application to the computation of assembly behaviours is different. We do not consider a composition of processes as an environment of some fixed process but analyse instead the behavioural impact of leaf components. This allows, on the one hand, to obtain a reduction algorithm generating an assembly with less components that may be used to substitute the original assembly. On the other hand, we may improve the efficiency of the neutrality analysis by considering port instead of component behaviours.

4.2. Neutrality and PADL compatibility. PADL [**BCD02**] uses observational equivalence for a notion of compatibility between components that expresses the same idea as our notion of behavioural neutrality. In PADL compatibility is used to detect architectural mismatches and it is shown that pairwise compatibility is a sufficient criterion to derive deadlock-freedom of an acyclic assembly from the deadlock-freedom of its local components. This technique is, however, not applied for an explicit construction of syntactically reduced assemblies. The syntactic reduction has the advantage that we can abstract from synchronisations within an assembly by removing neutral components stepwise. It allows us to utilise again behavioural neutrality after each of the reduction steps. By this means we finally obtain a reduced assembly whose behaviour is known to be equivalent to the original behaviour of the particular composite component.

Finally, deciding neutrality of component behaviours can still be quite expensive due to the required check for greybox equivalence. Therefore, as discussed above in the comparison with [**CK96**], a further important difference between our approach and [**BCD02**] concerns the integration of port behaviours. This increases the efficiency of the check for neutrality, under the assumption of weak determinism.

CHAPTER 4

A Theory for Refinement and Compatibility

1.	**Input-Persistent I/O-Transition Systems (PIO)**	**48**
1.1.	From IOTSs to PIOs to MIOs	48
1.2.	Blackbox-Refinement	50
1.3.	Output Compatibility	55
2.	**Asynchronous Communication**	**58**
2.1.	FIFO Encoding and Buffered PIOs	59
2.2.	Refinement and Asynchronous Compatibility	61
3.	**Compatibility and N-ary Composition**	**64**
3.1.	Communication-Safety	64
3.2.	Pairwise and Incremental Analysis	65
4.	**Greybox Refinement and General Properties**	**66**
4.1.	Blackbox vs Greybox Refinement	69
4.2.	Weak Simulation and Temporal Logics	69
5.	**Discussion and Related Work**	**73**

In order to support the development and analysis of component behaviours on different abstraction levels, a theory for the refinement of behaviour specifications is needed. The refinement relation should satisfy fundamental properties such as reflexivity, transitivity, and compositionality. Moreover, the relation should, on the one hand, preserve as much properties as possible, but on the other hand leave enough freedom for concrete design decisions. Our study is motivated by the aim to detect behavioural incompatibilities as discussed in Chap. 2, taking into account the asymmetry of sending (output) and receiving (input) messages. We develop different notions of output compatibility with varying degrees of freedom for the interleaving of local, non-shared component actions. It turns out that, in order to preserve output compatibility, we need additional information that allows to couple the message transfer with the refinement of the transition that specified the transfer. On this account our approach extends I/O-transition systems to input-persistent I/O-transition systems (PIO) and defines a blackbox refinement relation that demands the preservation of persistent transitions. The relation is termed "blackbox" because it ignores differences of internal labels. PIOs can be considered as a special case of modal I/O automata (MIO) and then, blackbox refinement corresponds to weak modal refinement.

Our theory is required to cope with FIFO buffered message exchange. Based on an encoding of FIFO buffers we show that blackbox refinement is indeed transferable to asynchronously communicating components. Moreover, we develop a notion of asynchronous output compatibility that is a natural extension of synchronous output compatibility to an asynchronous setting and show its preservation by blackbox refinement. The resulting theory is an interface theory in the sense of [**BMSH10**] which is briefly discussed in Sect. 5.

Communication-safety defines a notion of output compatibility for n-ary compositions of PIOs. We show that blackbox refinement of single transition systems preserves communication-safety of their composition and, moreover, investigate to what extend pairwise analysis of output compatibility implies communication-safety.

Finally, we consider more general properties and a greybox variant of our refinement relation. Greybox refinement distinguishes internal labels. We show that greybox refinement preserves safety properties with regard to the communication traces of a composed system and briefly discuss counterexamples for the preservation of liveness properties.

1. Input-Persistent I/O-Transition Systems (PIO)

We extend the formalism of I/O-transition systems given in Chap. 2 (Sect. 2) to include the specification of a distinguished set of transitions, so-called persistent transitions. These kinds of transitions define additional information for a subset of the transitions given by the transition relation of an I/O-transition system. The additional information is used to relate notions of compatibility with a refinement relation for input-persistent I/O-transition systems in such a way that refinement preserves compatibility and is still compositional without assumptions on the environment of the given transition system.

1.1. From IOTSs to PIOs to MIOs. The overall goal is to develop a theory for port-based components which communicate by synchronous and asynchronous message exchange. In such a setting, we expect component implementations to support at least all of the specified input behaviour, i.e. to provide guarantees for the reception of messages. For this reason, we consider a particular class of IOTSs with persistent transitions, namely those which are input-persistent. The corresponding refinement relation is then defined in such a way that persistent transitions, and thus the input behaviour according to a corresponding specification, is preserved.

Definition 4.1 (Input-persistent I/O-transition system) *An* input-persistent I/O-transition system (PIO)*, is a tuple* (L, S, s_0, Δ, Π)*, where* (L, S, s_0, Δ) *is an I/O-transition system and* Π *is a set of* persistent transitions*, given by* $\Pi \subseteq \Delta$ *such that for* $L = (I, O, T)$*, for all* $l \in I$ *and all* $s, s' \in S$*, if* $(s, l, s') \in \Delta$ *then* $(s, l, s') \in \Pi$.

The definition and notations for IOTSs are extended in a straightforward way for PIOs. For instance, if $A = (L, S, s_0, \Delta, \Pi)$ is a PIO, we continue to use an index to refer to the constituents of A, e.g. L_A to refer to the IOL L, Π_A to refer to Π etc. An extension of the operators of I/O-transition systems which takes the set of persistent transitions into account yields the corresponding operators of PIOs. We assume that such a transfer preserves the assignment of transitions to the relations Δ and Π. For instance, we will use the ξ operator to hide internal labels in PIOs; then it is assumed that ξ is applied to both relations and the hidden transitions are still in the respective relation.

In the following we give an explicit definition for composability and the product operator. The product preserves persistence of a transition only in case both transitions are persistent. Two PIOs A and B are *composable* if L_A and L_B are composable. The *product* of two composable PIOs A and B is the PIO $A \otimes B = (L_A \otimes L_B, S_A \times S_B, (s_{0,A}, s_{0,B}), \Delta, \Pi)$ where Δ and Π are given for $R \in \{\Delta, \Pi\}$ by

$$\begin{aligned} R = \{((s_A, s_B), l, (s'_A, s_B)) \mid (s_A, l, s'_A) \in R_A \wedge s_B \in S_B \wedge l \notin \mathcal{S}(A, B)\} \cup \\ \{((s_A, s_B), l, (s_A, s'_B)) \mid (s_B, l, s'_B) \in R_B \wedge s_A \in S_A \wedge l \notin \mathcal{S}(A, B)\} \cup \\ \{((s_A, s_B), l, (s'_A, s'_B)) \mid (s_A, l, s'_A) \in R_A \wedge (s_B, l, s'_B) \in R_B \wedge l \in \mathcal{S}(A, B)\}. \end{aligned}$$

Example 4.2 (PIO Composition) *Figure 4.1 shows a simple system comprising a producer process* Prod *and a buffer process* Buf*. If we assume that* put *is the sole shared action, the composition results in a PIO as depicted in Fig. 4.1c. Note that the distinction between* Δ *and persistent transitions in* Π *does not play a special role for the composition of PIOs. Transitions with non-shared labels are interleaved in the product and transitions with shared labels synchronise by a rendezvous mechanism. Though, note that synchronisation requires the transitions to be from the same relation, that is, persistent transitions*

must be specified to be persistent in both systems. If this is not the case the transition is synchronised in Δ but not in the set of persistent transitions of the product.

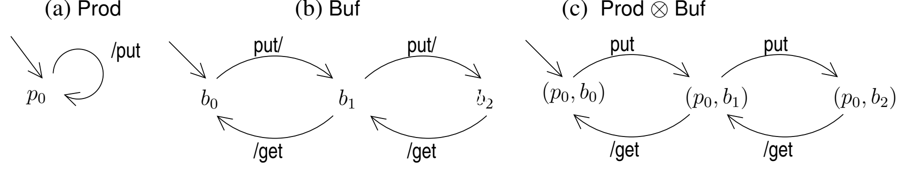


FIGURE 4.1. PIOs for producer and buffer

Within this chapter we will make extensive use of the notation Δ^{X}_A introduced in Sect. 2.1 of Chap. 2 to abbreviate conditions for skipping transitions in the transition relation of an IOTS. Therefore, we recall the initial definition by an informal transfer to PIOs. Let A be a PIO, $\mathrm{X} \subseteq \mathcal{L}(A)$ and $R \in \{\Delta, \Pi\}$. Then $(s, s') \in R^{\mathrm{X}}_A$ means s' is reachable in R using labels in X, and $(s, l, s') \in R^{\mathrm{X}}_A$ means that a transition $(\bar{s}, l, \bar{s'})$ exists in R which is preceeded and succeeded by transitions using labels in X, that is, $(s, \bar{s}) \in R^{\mathrm{X}}_A$ and $(\bar{s}', s') \in R^{\mathrm{X}}_A$.

The following lemma formalises a basic property concerning the reachability of global states in the composition of PIOs along paths of non-shared labels.

Lemma 4.3 (Non-shared reachability) *Let A and B be composable PIOs and let X be a set of labels such that $\mathrm{X} \cap \mathcal{S}(A, B) = \emptyset$. If $(s_A, s_B) \in \mathscr{R}(A \otimes B)$, then for all $l_A \in \mathcal{L}(A)$, $l \in \mathcal{L}(A) \cup \mathcal{L}(B)$ such that $l_A \notin \mathcal{S}(A, B) \cup \mathrm{X}$ and $l \in \mathcal{S}(A, B)$ the following holds for $R \in \{\Delta, \Pi\}$:*

(1) $\quad (s_A, s'_A) \in R^{\mathrm{X}}_A \implies ((s_A, s_B), (s'_A, s_B)) \in R^{\mathrm{X}}_{A\otimes B}$,

(2) $\quad (s_A, l_A, s'_A) \in R^{\mathrm{X}}_A \implies ((s_A, s_B), l_A, (s'_A, s_B)) \in R^{\mathrm{X}}_{A\otimes B}$,

(3) $\quad (s_A, l, s'_A) \in R^{\mathrm{X}}_A \wedge (s_B, l, s'_B) \in R_B \implies ((s_A, s_B), l, (s'_A, s'_B)) \in R^{\mathrm{X}}_{A\otimes B}$.

PROOF. (1) Since X consists of non-shared labels only, we can construct an execution fragment from (s_A, s_B) to (s'_A, s_B) using only transitions from R_A. (2) and (3) is shown using (1) before and after the transition labelled l_A, l respectively. □

PIOs in turn are an instance of consistent modal I/O automata with Δ the may- and Π the must-transition relation [**LNW07**]. In the succeeding sections we refer to modal I/O automata (MIO) as considered in [**BMSH10**] and relate some of our definitions and results by a translation $\mathrm{mio} : \mathrm{PIO} \to \mathrm{MIO}$ which maps the PIO-constituents to the corresponding elements of a MIO. The translation is straightforward, if we assume that all τ-transitions of a PIO have been relabelled to internal transitions with new internal labels in T before. The *mio-translation of a τ-free PIO* $((I, O, T), S, s_0, \Delta, \Pi)$ is given by

$$\mathrm{mio}((I, O, T), S, s_0, \Delta, \Pi) = (\mathit{states}, \mathit{start}, (\mathit{in}, \mathit{out}, \mathit{int}), \dashrightarrow, \longrightarrow), \text{ such that}$$

$\mathit{states} = S$, $\mathit{start} = s_0$, $(\mathit{in}, \mathit{out}, \mathit{int}) = (I, O, T)$, and $(s, l, s') \in \dashrightarrow$ iff $(s, l, s') \in \Delta$, and $(s, l, s') \in \longrightarrow$ iff $(s, l, s') \in \Pi$. The extension of the IOTS product operator for PIOs mentioned above coincides with the product operator as defined for modal I/O automata. We expect the remaining operators to coincide analogously.

The difference between PIOs and MIOs lies in the requirement of input-persistence, disallowing what is called "may input" for modal I/O automata. From a technical point of view anything which is developed for PIOs within this chapter could be developed equally for MIOs. Our conceptual departure, however, aims at a theory with support for modular refinement while preserving a notion of compatibility which guarantees the absence of certain communication errors. As argued in the following, the additional expressive power of MIOs does not contribute in the given setting.

Consider the example in Fig. 4.2 as a specification of a two-element buffer. The first input is a so-called "must transition" and the second input is a so-called "may transition" (indicated by the dashed line) and thus, in terms of weak modal refinement, the given MIO is refinable by either an one-element or a two-element buffer.

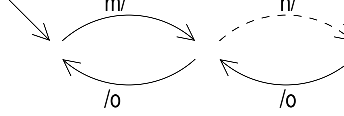


FIGURE 4.2. Modal I/O automata with may input (dashed line)

Since PIOs do not allow for "may input transitions" the specification in Fig. 4.2 can not be expressed using PIOs. But a closer look reveals that may inputs are only of limited use in our setting of output compatible environments. A communication partner on the same level of abstraction can not send the second input, since the given MIO would not be compatible due to the "may" attribute of the given transition. Moreover, MIO refinement does not allow to add output transitions in the more concrete transition system. Therefore, the given may input is neither directly usable nor by potentially refined communication partners. Instead, communication partners using the may input must be specified with respect to a refinement of the given MIO which guarantees the existence of the second input transition by tagging it as "must input transition". In general this means that in order to make an input usable in output compatible contexts we need to use "must inputs", and this is exactly our requirement of input persistence in the definition of PIOs.

1.2. Blackbox-Refinement. Even though we do not consider components with data states, refinement is based on an intuitive understanding as known from correctness relations for operation specifications in the form of pre- and postconditions. A correct implementation should work properly at least in all states where the given precondition holds, and in this case should establish the postcondition, i.e., reach at most one of the states where the given postcondition holds. This co-/contravariant interpretation of behavioural specifications can be applied to transition systems with inputs and outputs too. A correct implementation should support at least the given inputs and use at most the specified outputs. Additionally arbitrary internal actions may be used, as long as correctness w.r.t. inputs and outputs is not touched.

A direct transfer of such an interpretation, however, bears problems concerning the preservation of our notion of compatibility, motivated in Chap. 2 (Sect. 1.3) and formally defined below. Therefore, the given basic intuition is combined with the notion of persistent transitions in the following way: Specified inputs must always exist in the refined transition system (R1). At most the specified outputs may exist in the concrete system (R2). Specified internal transitions may be skipped and concrete internal transitions may be added as long as the refinement relation is not invalidated (R3). Additionally, persistent transitions are mandatory to be taken into account within a correct refinement (R4). A correct implementation is obliged to preserve persistent output transitions; persistent internal steps need to be mimicked by arbitrary internal steps of the concrete system (including doing no step at all) such that the refinement relation is not invalidated. Finally, we do not make a difference in the treatment of internal and τ transitions. By this means the relation can be used equivalently for systems which either show internal transitions or did hide such transitions by a relabelling to the anonymous action τ already.

Definition 4.4 (Blackbox refinement) *Let A and C be PIOs. C is a* blackbox refinement *of A, written $C \sqsubseteq^{\mathrm{bb}} A$, if $I_A = I_C$, $O_A = O_C$ and there exists $R \subseteq S_A \times S_C$ such that $(s_{0,A}, s_{0,C}) \in R$ and for all $(s_A, s_C) \in R$ the conditions in Tab. 4.3 hold.*

(A-I/O)	$\forall l \in I_A \cup O_A \;.\; \forall s'_A \in S_A \;.\; (s_A, l, s'_A) \in \Pi_A \implies$
	$(\exists s'_C \in S_C \;.\; (s_C, l, s'_C) \in \Pi_C^{\mathcal{T}(C)} \wedge (s'_A, s'_C) \in R)$
(A-Int)	$\forall l \in \mathcal{T}(A) \;.\; \forall s'_A \in S_A \;.\; (s_A, l, s'_A) \in \Pi_A \implies$
	$(\exists s'_C \in S_C \;.\; (s_C, s'_C) \in \Pi_C^{\mathcal{T}(C)} \wedge (s'_A, s'_C) \in R)$
(C-I/O)	$\forall l \in I_C \cup O_C \;.\; \forall s'_C \in S_C \;.\; (s_C, l, s'_C) \in \Delta_C \implies$
	$(\exists s'_A \in S_A \;.\; (s_A, l, s'_A) \in \Delta_A^{\mathcal{T}(A)} \wedge (s'_A, s'_C) \in R)$
(C-Int)	$\forall l \in \mathcal{T}(C) \;.\; \forall s'_C \in S_C \;.\; (s_C, l, s'_C) \in \Delta_C \implies$
	$(\exists s'_A \in S_A \;.\; (s_A, s'_A) \in \Delta_A^{\mathcal{T}(A)} \wedge (s'_A, s'_C) \in R)$

FIGURE 4.3. Conditions for blackbox refinement between A and C

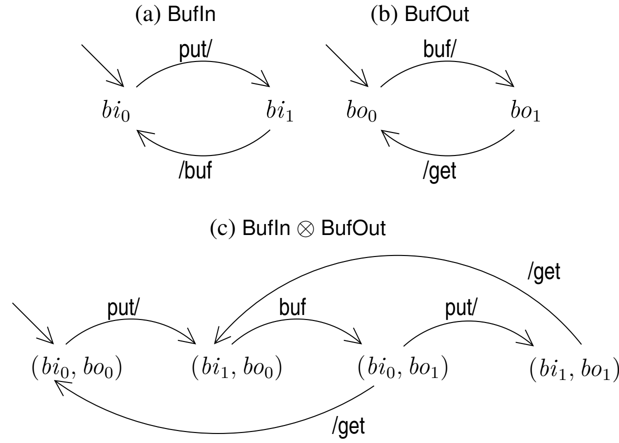


FIGURE 4.4. Two one-element buffers and their composition

Since we consider input-persistent I/O-transition systems the requirement R1 is taken into account by condition (A-I/O). Requirement R2 is modelled by (C-I/O), and the requirements for internal transitions motivate the application of $\Delta_A^{\mathcal{T}(A)}$ in conditions (C-I/O) and (C-Int). The requirement for persistent output and internal transitions is included in (A-I/O) and taken into account by (A-Int) respectively.

Hiding internal labels does not invalidate a given blackbox refinement due to conditions (A-Int) and (C-Int) which do not distinguish internal and τ transitions.

Lemma 4.5 *Let A and C be PIOs. If $C \sqsubseteq^{\mathrm{bb}} A$ then $C \sqsubseteq^{\mathrm{bb}} A\xi$, $C\xi \sqsubseteq^{\mathrm{bb}} A$, and $C\xi \sqsubseteq^{\mathrm{bb}} A\xi$.*

Example 4.6 (Blackbox refinement) *Assume we would like to provide an implementation for Buf (Fig. 4.1) using the composition of two one-element buffers as depicted in Fig. 4.4. Then, the composition of BufIn and BufOut should be in a refinement relation with Buf. Indeed, we have BufIn $\otimes$ BufOut $\sqsubseteq^{\mathrm{bb}}$ Buf, which is witnessed by the relation $R \subseteq S_{\mathit{Buf}} \times S_{\mathit{BufIn} \otimes \mathit{BufOut}}$ with $R = \{(b_0, (bi_0, bo_0)), (b_1, (bi_1, bo_0)), (b_1, (bi_0, bo_1)), (b_2, (bi_1, bo_1))\}$.*

The definition of blackbox refinement coincides with weak modal refinement of modal I/O-automata if we consider PIOs with an identical set of internal labels.

Proposition 4.7 (Weak modal refinement) *Let A and C be PIOs with* $T_C = T_A$. $\mathrm{mio}(C) \preceq^*_{\mathrm{m}} \mathrm{mio}(A)$ *if, and only if* $C \sqsubseteq^{\mathrm{bb}} A$.

PROOF. The notation $\rightarrow^*_{\mathrm{mio}(A)}$ for internal paths used in [**BMSH10**] and the notation Δ^{X} used here can be shown to be equivalent. Thus the conditions (1) and (2) of weak modal refinement in [**BMSH10**] can be separated to conditions (1a), (1b), (2a) and (2b) such that (1a) is equivalent to (A-I/O), (1b) to (A-Int), (2a) to (C-I/O) and (2b) to (C-Int).

Our mio-translation requires τ-free PIOs since modal I/O-automata allow for internal labels but do not include τ labels. In the following we assume that τ labels of A and C have been relabelled appropriately to new internal labels before. We use t as a generic identifier of an internal label. Let $\mathrm{mio}(A) = (\mathit{states}_A, \mathit{start}_A, (\mathit{in}_A, \mathit{out}_A, \mathit{int}_A), \dashrightarrow_A, \longrightarrow_A)$ and analogous for $\mathrm{mio}(C)$. Let $R_{CA} \subseteq \mathit{states}_C \times \mathit{states}_A$ be a witness for $\mathrm{mio}(C) \preceq^*_{\mathrm{m}} \mathrm{mio}(A)$. Then clause (1) of weak modal refinement reads as follows: for all $(s_C, s_A) \in R_{CA}$ it holds that

$$\forall s'_A \in \mathit{states}_A \,.\, (s_A, l, s'_A) \in \longrightarrow_{\mathrm{mio}(A)} \implies (\exists s'_C \in \mathit{states}_C \,.$$
$$(((s_C, l, s'_C) \in \longrightarrow^*_{\mathrm{mio}(C)} \wedge l \notin \mathit{int}_C) \vee (s_C, t, s'_C) \in \longrightarrow^*_{\mathrm{mio}(C)}) \wedge (s'_C, s'_A) \in R_{CA})$$

The conditions for $\mathrm{mio}(C)$ can be separated:

(1a) $\forall l \in \mathit{ext}_A \,.\, \forall s'_A \in \mathit{states}_A \,.\, (s_A, l, s'_A) \in \longrightarrow_{\mathrm{mio}(A)} \implies (\exists s'_C \in \mathit{states}_C \,.$
$(s_C, l, s'_C) \in \longrightarrow^*_{\mathrm{mio}(C)} \wedge (s'_C, s'_A) \in R_{CA})$

(1b) $\forall l \in \mathit{int}_A \,.\, \forall s'_A \in \mathit{states}_A \,.\, (s_A, l, s'_A) \in \longrightarrow_{\mathrm{mio}(A)} \implies (\exists s'_C \in \mathit{states}_C \,.$
$(s_C, t, s'_C) \in \longrightarrow^*_{\mathrm{mio}(C)} \wedge (s'_C, s'_A) \in R_{CA})$

Consider clause (1a). By definition, $(s_C, l, s'_C) \in \longrightarrow^*_{\mathrm{mio}(C)}$ iff there are $\bar{s}_C, \bar{s}'_C \in \mathit{states}_C$ such that $(s_C, t, \bar{s}_C) \in \longrightarrow^*_{\mathrm{mio}(C)}$, $(\bar{s}_C, l, \bar{s}'_C) \in \longrightarrow_{\mathrm{mio}(C)}$ and $(\bar{s}'_C, t, s'_C) \in \longrightarrow^*_{\mathrm{mio}(C)}$, which is equivalent to $(s_C, t, \bar{s}_C) \in \Pi^{\mathcal{T}(C)}_C$, $(\bar{s}_C, l, \bar{s}'_C) \in \Pi_C$ and $(\bar{s}'_C, t, s'_C) \in \Pi^{\mathcal{T}(C)}_C$. Hence $(s_C, l, s'_C) \in \Pi^{\mathcal{T}(C)}_C$ and thus (1a) translates to

$$\forall l \in \mathit{in}_A \cup \mathit{out}_A \,.\, \forall s'_A \in \mathit{states}_A \,.\, (s_A, l, s'_A) \in \Pi_A \implies (\exists s'_C \in \mathit{states}_C \,.$$
$$(s_C, l, s'_C) \in \Pi^{\mathcal{T}(C)}_C \wedge (s'_C, s'_A) \in R_{CA}),$$

which corresponds to clause (A-I/O) of blackbox refinement. Condition (1b) and condition (2) for may-transitions are translated analogously. □

Satisfying the requirements of a preorder is a natural demand for refinement relations between specifications of component behaviour. In particular transitivity is important, as one usually aims to support development processes based on stepwise refinement. Relations which allow for the removal of abstract transitions in the concrete transition system sometimes are at risk to invalidate this property. Though, as shown in the following this is not the case with blackbox refinement.

Proposition 4.8 (Preorder) *Let A, C and D be PIOs. Then* $A \sqsubseteq^{\mathrm{bb}} A$ *and if* $D \sqsubseteq^{\mathrm{bb}} C$ *and* $C \sqsubseteq^{\mathrm{bb}} A$ *then* $D \sqsubseteq^{\mathrm{bb}} A$.

The proof for transitivity uses the following Lemma.

Lemma 4.9 *Let* R_{AC} *be a witness for* $C \sqsubseteq^{\mathrm{bb}} A$. *Let* $(s_A, s_C) \in R_{AC}$ *and assume* $l \in I_A \cup O_A$. *Then*

(1) $(s_A, s'_A) \in \Pi^{\mathcal{T}(A)}_A \implies \exists s'_C \in S_C \,.\, (s_C, s'_C) \in \Pi^{\mathcal{T}(C)}_C \wedge (s'_A, s'_C) \in R_{AC}$
(2) $(s_A, l, s'_A) \in \Pi^{\mathcal{T}(A)}_A \implies \exists s'_C \in S_C \,.\, (s_C, l, s'_C) \in \Pi^{\mathcal{T}(C)}_C \wedge (s'_A, s'_C) \in R_{AC}$
(3) $(s_C, s'_C) \in \Delta^{\mathcal{T}(C)}_C \implies \exists s'_A \in S_A \,.\, (s_A, s'_A) \in \Delta^{\mathcal{T}(A)}_A \wedge (s'_A, s'_C) \in R_{AC}$
(4) $(s_C, l, s'_C) \in \Delta^{\mathcal{T}(C)}_C \implies \exists s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta^{\mathcal{T}(A)}_A \wedge (s'_A, s'_C) \in R_{AC}$

PROOF OF LEMMA. Consider (2). By $(s_A, l, s'_A) \in \Pi_A^{\mathcal{T}(A)}$ there exists $l_{A,i} \in \mathcal{T}(A)$ and $s_{A,i} \in S_A$ for $1 \leq i < n$ such that $(s_A, l_{A,1}, s_{A,1}) \in \Pi_A$, $(s_{A,i}, l_{A,i+1}, s_{A,i+1}) \in \Pi_A$, ..., $(s_{A,n}, l, s'_A) \in \Pi_A$. By induction on the fragment length n and using R_{AC} there exists $s_{C,i}, s'_C \in S_C$ such that $(s_C, l_{A,1}, s_{C,1}) \in \Pi_C$, $(s_{C,i}, l_{A,i+1}, s_{C,i+1}) \in \Pi_C$, ..., $(s_{C,n}, l, s'_C) \in \Pi_C$ and $(s_{A,i}, s_{C,i}) \in R_{AC}$ for all $1 \leq i < n$ and $(s'_A, s'_C) \in R_{AC}$. The remaining cases hold analogously. □

PROOF OF PROPOSITION. Reflexivity holds by using the identity relation as a witness for $A \sqsubseteq^{\mathrm{bb}} A$. Transitivity is shown using $R = \{(s_A, s_D) \mid \exists s_C \in S_C \,.\, (s_A, s_C) \in R_{AC} \wedge (s_C, s_D) \in R_{CD}\}$, where R_{AC} and R_{CD} are assumed to witness $C \sqsubseteq^{\mathrm{bb}} A$ and $D \sqsubseteq^{\mathrm{bb}} C$ respectively. Obviously $(s_{A,0}, s_{D,0}) \in R$. Let $(s_A, s_D) \in R$ and let $s_C \in S_C$ such that $(s_A, s_C) \in R_{AC}$ and $(s_C, s_D) \in R_{CD}$. We show that R satisfies the conditions of black box refinement.

(Case A-I/O) Let $l \in I_A \cup O_A$ and $(s_A, l, s'_A) \in \Pi_A$. By (A-I/O) for R_{AC} there exists $s'_C \in S_C$ such that $(s_C, l, s'_C) \in \Pi_C^{\mathcal{T}(C)}$ and $(s'_A, s'_C) \in R_{AC}$. By Lem. 4.9 for R_{CD} and $I_A = I_C = I_D$ and $O_A = O_C = O_D$, there exists $s'_D \in S_D$ such that $(s_D, l, s'_D) \in \Pi_D^{\mathcal{T}(D)}$ and $(s'_C, s'_D) \in R_{CD}$ and hence $(s'_A, s'_D) \in R$.

(Case A-Int) Let $l \in \mathcal{T}(A)$ and $(s_A, l, s'_A) \in \Pi_A$. By (A-Int) for R_{AC} there exists $s'_C \in S_C$ such that $(s_C, s'_C) \in \Pi_C^{\mathcal{T}(C)}$ and $(s'_A, s'_C) \in R_{AC}$. By Lem. 4.9 for R_{CD} there exists $s'_D \in S_D$ such that $(s_D, s'_D) \in \Pi_D^{\mathcal{T}(D)}$ and $(s'_C, s'_D) \in R_{CD}$ and hence $(s'_A, s'_D) \in R$.

(Case C-I/O) Let $l \in I_D \cup O_D$ and $(s_D, l, s'_D) \in \Delta_D$. By (C-I/O) for R_{CD} there exists $s'_C \in S_C$ such that $(s_C, l, s'_C) \in \Delta_C^{\mathcal{T}(C)}$ and $(s'_C, s'_D) \in R_{CD}$. By Lem. 4.9 for R_{AC} and $I_A = I_C = I_D$ and $O_A = O_C = O_D$, there exists $s'_A \in S_A$ such that $(s_A, l, s'_A) \in \Delta_A^{\mathcal{T}(A)}$ and $(s'_A, s'_C) \in R_{AC}$ and hence $(s'_A, s'_D) \in R$.

(Case C-Int) Let $l \in \mathcal{T}(D)$ and $(s_D, l, s'_D) \in \Delta_D$. By (C-Int) for R_{CD} there exists $s'_C \in S_C$ such that $(s_C, s'_C) \in \Delta_C^{\mathcal{T}(C)}$ and $(s'_C, s'_D) \in R_{CD}$. By Lem. 4.9 for R_{AC} there exists $s'_A \in S_A$ such that $(s_A, s'_A) \in \Delta_A^{\mathcal{T}(A)}$ and $(s'_A, s'_C) \in R_{AC}$ and hence $(s'_A, s'_D) \in R$. □

Another fundamental property of any refinement relation is precongruence with respect to the product operator. Precongruence allows for modular substitution of abstract transition systems by a refinement such that the resulting composition is a refinement of the original composition. For the case of PIOs we assume composability of the given transition systems. Composability represents our syntactical requirements on the proper composition of communication partner: outputs should be provided by the particular partner and internal labels should be disjoint from each other, such that these transitions indeed represent internal, local actions of the given process.

Theorem 4.10 (Precongruence) *Let A, B and C be PIOs such that A, B and C, B are composable. If $C \sqsubseteq^{\mathrm{bb}} A$, then $C \otimes B \sqsubseteq^{\mathrm{bb}} A \otimes B$.*

PROOF. Let $A \otimes B = ((I_{AB}, O_{AB}, T_{AB}), S_{AB}, s_{0,AB}, \Delta_{AB}, \Pi_{AB})$ and $C \otimes B = ((I_{CB}, O_{CB}, T_{CB}), S_{CB}, s_{0,CB}, \Delta_{CB}, \Pi_{CB})$. Let R_{AC} be a witness for $C \sqsubseteq^{\mathrm{bb}} A$ with $(s_{0,A}, s_{0,C}) \in R_{AC}$ and let

$$R = \{((s_A, s_B), (s_C, s_B)) \mid (s_A, s_C) \in R_{AC} \wedge (s_A, s_B) \in \mathscr{R}(A \otimes B)\},$$

then $((s_{0,A}, s_{0,B}), (s_{0,C}, s_{0,B})) \in R$. Let $((s_A, s_B), (s_C, s_B)) \in R$. We check the conditions of blackbox refinement for R.

(Case A-I/O) Let $l \in I_{AB} \cup O_{AB}$ and $((s_A, s_B), l, (s'_A, s'_B)) \in \Pi_{AB}$.
If $l \in I_A \cup O_A$, then there exists $(s_A, l, s'_A) \in \Pi_A$ and $s_B = s'_B$. By clause (A-I/O) for R_{AC} there exists $s'_C \in S_C$ such that $(s_C, l, s'_C) \in \Pi_C^{\mathcal{T}(C)}$ and $(s'_A, s'_C) \in R_{AC}$. Since $l \notin \mathcal{S}(C, B)$ (by $l \notin \mathcal{S}(A, B)$, $I_A = I_C$ and $O_A = O_C$) we have by Lem. 4.3 (2) and

$\mathcal{T}(C) \subseteq \mathcal{T}(CB)$, that $((s_C, s_B), l, (s'_C, s_B)) \in \Pi_{CB}^{\mathcal{T}(CB)}$. Since $(s'_A, s_B) \in \mathscr{R}(A \otimes B)$ it follows that $((s'_A, s_B), (s'_C, s_B)) \in R$.
If $l \in I_B \cup O_B$, then there exists $(s_B, l, s'_B) \in \Pi_B$ and $s_A = s'_A$. Hence $((s_C, s_B), l, (s_C, s'_B)) \in \Pi_{CB}$ and since $(s_A, s'_B) \in \mathscr{R}(A \otimes B)$, it follows that $((s_A, s'_B), (s_C, s'_B)) \in R$.

(Case A-Int) Let $l \in \mathcal{T}_{AB}$ and $((s_A, s_B), l, (s'_A, s'_B)) \in \Pi_{AB}$.
If $l \in \mathcal{T}(A)$, then there exists $(s_A, l, s'_A) \in \Pi_A$ and $s_B = s'_B$. By (A-Int) for R_{AC} there exists $s'_C \in S_C$ such that $(s_C, s'_C) \in \Pi_C^{\mathcal{T}(C)}$ and $(s'_A, s'_C) \in R_{AC}$. Since $l \notin \mathcal{S}(C, B)$ (by $l \notin \mathcal{S}(A, B)$, $I_A = I_C$, $O_A = O_C$ and C, B composable), we have by Lem. 4.3 (1) and $\mathcal{T}(C) \subseteq \mathcal{T}(CB)$, that $((s_C, s_B), (s'_C, s_B)) \in \Pi_{CB}^{\mathcal{T}(CB)}$. Since $(s'_A, s_B) \in \mathscr{R}(A \otimes B)$ it follows that $((s'_A, s_B), (s'_C, s_B)) \in R$.
If $l \in \mathcal{T}(B)$, then there exists $(s_B, l, s'_B) \in \Pi_B$ and $s_A = s'_A$. Since $l \notin \mathcal{S}(C, B)$ (by composability of C and B), we have $((s_C, s_B), l, (s_C, s'_B)) \in \Pi_{CB}$ and since $(s_A, s'_B) \in \mathscr{R}(A \otimes B)$ it follows that $((s_A, s'_B), (s_C, s'_B)) \in R$.
If $l \in \mathcal{S}(A, B)$, then there exists $(s_A, l, s'_A) \in \Pi_A$ and $(s_B, l, s'_B) \in \Pi_B$, and either $l \in I_A \cap O_B$ or $l \in O_A \cap I_B$. In both cases, by (A-I/O) for R_{AC}, there exists $s'_C \in S_C$ such that $(s_C, l, s'_C) \in \Pi_C^{\mathcal{T}(C)}$ and $(s'_A, s'_C) \in R_{AC}$. By Lem. 4.3 (3) and $\mathcal{T}(C) \subseteq \mathcal{T}(CB)$ it follows that $((s_C, s_B), l, (s'_C, s'_B)) \in \Pi_{CB}^{\mathcal{T}(CB)}$ and since $(s'_A, s'_B) \in \mathscr{R}(A \otimes B)$ we have $((s'_A, s'_B), (s'_C, s'_B)) \in R$.

(Case C-I/O) Let $l \in I_{CB} \cup O_{CB}$ and $((s_C, s_B), l, (s'_C, s'_B)) \in \Pi_{CB}$.
If $l \in I_C \cup O_C$, then there exists $(s_C, l, s'_C) \in \Delta_C$ and $s_B = s'_B$. By clause (C-I/O) for R_{AC} there exists $s'_A \in S_A$ such that $(s_A, l, s'_A) \in \Delta_A^{\mathcal{T}(A)}$ and $(s'_A, s'_C) \in R_{AC}$. By Lem. 4.3 (1) and $\mathcal{T}(A) \subseteq \mathcal{T}(AB)$ it follows that $((s_A, s_B), l, (s'_A, s_B)) \in \Delta_{AB}^{\mathcal{T}(AB)}$. Since $(s'_A, s_B) \in \mathscr{R}(A \otimes B)$ we also have $((s'_A, s_B), (s'_C, s_B)) \in R$.
If $l \in I_B \cup O_B$, then there exists $(s_B, l, s'_B) \in \Delta_B$ and $s_C = s'_C$. Hence $((s_A, s_B), l, (s_A, s'_B)) \in \Delta_{AB}$ and since $(s_A, s'_B) \in \mathscr{R}(A \otimes B)$ and $(s_A, s_C) \in R_{AC}$ it follows that $((s_A, s'_B), (s_C, s'_B)) \in R$.

(Case C-Int) Let $l \in \mathcal{T}_{CB}$ and $((s_C, s_B), l, (s'_C, s'_B)) \in \Delta_{CB}$.
If $l \in \mathcal{T}(C)$, then there exists $(s_C, l, s'_C) \in \Delta_C$ and $s_B = s'_B$. By (C-Int) for R_{AC} there exists $s'_A \in S_A$ such that $(s_A, s'_A) \in \Delta_A^{\mathcal{T}(A)}$ and $(s'_A, s'_C) \in R_{AC}$. Since $l \notin \mathcal{S}(A, B)$ (by $l \in \mathcal{T}(C)$, $I_A = I_C$, $O_A = O_C$ and A, B composable), we have by Lem. 4.3 (1) and $\mathcal{T}(A) \subseteq \mathcal{T}(AB)$, that $((s_A, s_B), (s'_A, s_B)) \in \Delta_{AB}^{\mathcal{T}(AB)}$. Since $(s'_A, s_B) \in \mathscr{R}(A \otimes B)$ it follows that $((s'_A, s_B), (s'_C, s_B)) \in R$.
If $l \in \mathcal{T}(B)$, then there exists $(s_B, l, s'_B) \in \Delta_B$ and $s_C = s'_C$, we have $((s_A, s_B), l, (s_A, s'_B)) \in \Delta_{AB}$ and hence $(s_A, s'_B) \in \mathscr{R}(A \otimes B)$ (by composability of A and B and $l \notin \mathcal{S}(A, B)$) and since $(s_A, s_C) \in R_{AC}$ it follows that $((s_A, s'_B), (s_C, s'_B)) \in R$.
If $l \in \mathcal{S}(C, B)$, then there exists $(s_C, l, s'_C) \in \Delta_C$ and $(s_B, l, s'_B) \in \Delta_B$, and either $l \in I_C \cap O_B$ or $l \in O_C \cap I_B$. In both cases, by (C-I/O) for R_{AC}, there exists $s'_A \in S_A$ such that $(s_A, l, s'_A) \in \Delta_A^{\mathcal{T}(A)}$ and $(s'_A, s'_C) \in R_{AC}$. By Lem. 4.3 (3) and $\mathcal{T}(A) \subseteq \mathcal{T}(AB)$ it follows that $((s_A, s_B), (s'_A, s'_B)) \in \Delta_{AB}^{\mathcal{T}(AB)}$ and hence $(s'_A, s'_B) \in \mathscr{R}(A \otimes B)$ and $((s'_A, s'_B), (s'_C, s'_B)) \in R$. □

Note that composability of C and B must be assumed due to the possibility of arbitrary additional internal labels in the concrete transition system.

Example 4.11 (Precongruence) *As discussed in Ex. 4.6 we have BufIn ⊗ BufOut $\sqsubseteq^{\mathrm{bb}}$ Buf, therefore we may replace Buf in the composition Prod ⊗ Buf and obtain by application of Thm. 4.10 a refinement of the original composition, that is Prod ⊗ (BufIn ⊗ BufOut) $\sqsubseteq^{\mathrm{bb}}$ Prod ⊗ Buf. Note that it is not required to hide synchronised transitions, since blackbox refinement makes no difference between τ and internal transitions.*

(A-Out) $\forall l \in O_A \cap I_B \,.\, \forall s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta_A \implies$
$(\exists s'_B \in S_B \,.\, (s_B, l, s'_B) \in \Pi_B^{\mathcal{O}_B})$

(B-Out) $\forall l \in O_B \cap I_A \,.\, \forall s'_B \in S_B \,.\, (s_B, l, s'_B) \in \Delta_B \implies$
$(\exists s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Pi_A^{\mathcal{O}_A})$

FIGURE 4.5. Output compatibility w.r.t. autonomous labels $\mathcal{O}_A$ and $\mathcal{O}_B$

Having shown that the given refinement relation is a preorder and a precongruence allows to conclude its compositionality. Therefore, properties which are preserved by blackbox refinement might be verified on the level of the composition of abstract transition systems and then we may conclude that the property still holds in a more concrete system if the abstract transition systems are replaced by proper refinements.

Corollary 4.12 (Compositionality) *Let A, B and C, D be PIOs such that A, B and C, D are composable. If $C \sqsubseteq^{\mathrm{bb}} A$ and $D \sqsubseteq^{\mathrm{bb}} B$, then $C \otimes D \sqsubseteq^{\mathrm{bb}} A \otimes B$.*

PROOF. By Thm. 4.10 and $C \sqsubseteq^{\mathrm{bb}} A$ we have $C \otimes B \sqsubseteq^{\mathrm{bb}} A \otimes B$. By Prop. 4.10 and $D \sqsubseteq^{\mathrm{bb}} B$ we have $C \otimes D \sqsubseteq^{\mathrm{bb}} C \otimes B$. Now, by Prop. 4.8, it follows that $C \otimes D \sqsubseteq^{\mathrm{bb}} A \otimes B$. □

1.3. Output Compatibility. The product operator for the composition of transition systems captures only successful communication. An I/O-transition without a matching counterpart does not occur in the composed transition system any more. In presence of distinguished input and output it becomes evident, that such a model is only appropriate for inputs (corresponding to unused provisions) but not for outputs (corresponding to required services), at least if one aims at some guarantees for message delivery. Therefore, a notion of output compatibility becomes an important issue. Within this section we consider three types of compatibility, strong, weak and ultra-weak output compatibility and analyse their preservation by blackbox refinement.

Definition 4.13 (Output compatibility) *Two composable PIOs A and B are* output compatible with respect to autonomous labels $\mathcal{O}_A \subseteq (O_A \setminus I_B) \cup \mathcal{T}(A)$ and $\mathcal{O}_B \subseteq (O_B \setminus I_A) \cup \mathcal{T}(B)$*, written $A \leftrightarrow_{(\mathcal{O}_A, \mathcal{O}_B)} B$, if for all $(s_A, s_B) \in \mathscr{R}(A \otimes B)$ the conditions in Fig. 4.5 hold.*

A and B are ultra-weakly output compatible*, written $A \leftrightarrow_u B$, if $\mathcal{O}_A = (O_A \setminus I_B) \cup \mathcal{T}(A)$ and $\mathcal{O}_B = (O_B \setminus I_A) \cup \mathcal{T}(B)$; A and B are* weakly output compatible*, written $A \leftrightarrow_w B$, if $\mathcal{O}_A = \mathcal{T}(A)$ and $\mathcal{O}_B = \mathcal{T}(B)$; A and B are* strongly output compatible*, written $A \leftrightarrow_s B$, if $\mathcal{O}_A = \mathcal{O}_B = \emptyset$.*

Weak output compatibility of PIOs corresponds to the notion of weak compatibility for MIOs defined in [**BMSH10**].

Example 4.14 (Output compatibility) *Consider the buffer PIO of Fig. 4.1 and assume $\{(b_1, \mathsf{get}, b_0), (b_2, \mathsf{get}, b_1)\} \subseteq \Pi_{\mathsf{Buf}}$, then* Prod *and* Buf *are ultra-weakly output compatible but neither weakly nor strongly output compatible. The interesting product state here is (p_0, b_2) which does not immediately provide a synchronising* put *input transition. Instead such an input is provided in state (p_0, b_1) after the unshared output transition* get*. Since we strictly assume output compatible environments such an unshared output will be synchronised later (if the source state is still reachable in the corresponding global state space) and hence we may skip such a transition while looking for a synchronising input.*

Similarly Prod *and* BufIn $\otimes$ BufOut *are ultra-weakly output compatible, if we assume that additionally to the* get*-transitions, the internal transition labelled* buf *is persistent, i.e. if $\Pi_{\mathsf{BufIn} \otimes \mathsf{BufOut}} \supseteq \{((bi_1, bo_1), \mathsf{get}, (bi_1, bo_0)),\ ((bi_0, bo_1), \mathsf{get}, (bi_0, bo_0)), ((bi_1, bo_0),$*

buf, $(bi_0, bo_1))\}$. *Here the interesting product state is* $(p_0, (bi_1, bo_0))$ *which shows an internal transition before the required input transition is reached. Internal transitions can be considered to be executable even without any environmental assumptions and hence also in this case, compatibility should not be at risk, if such a transition precedes a matching input.*

Weak and ultra-weak compatibility are preserved by blackbox refinement. Therefore, these notions of compatibility are examples of properties which are preserved if we replace abstract transition systems within a given composition with more concrete systems along blackbox refinement. First we consider ultra-weak output compatibility.

Proposition 4.15 (Ultra-weak preservation) *Let A, B and C be PIOs such that A, B and C, B are composable. If $A \leftrightarrow_u B$ and $C \sqsubseteq^{\mathrm{bb}} A$, then $C \leftrightarrow_u B$.*

The claim holds by the following reasoning which is detailed below. Let R_{AC} be a relation witnessing $C \sqsubseteq^{\mathrm{bb}} A$ and let $R = \{((s_A, s_B), (s_C, s_B)) \mid (s_A, s_C) \in R_{AC} \wedge (s_A, s_B) \in \mathscr{R}(A \otimes B)\}$. Lemma 4.16 (below) for the "upward" direction shows that, under the given assumptions, for any reachable state $(s_C, s_B) \in \mathscr{R}(C \otimes B)$ there is an $s_A \in S_A$ such that $(s_A, s_B) \in \mathscr{R}(A \otimes B)$ and $(s_A, s_C) \in R_{AC}$ and hence $((s_A, s_B), (s_C, s_B)) \in R$. Using (C-I/O) for R_{AC} any output of C can be related to an output of A and hence from $A \leftrightarrow_u B$ we obtain an input $(s_B, l, s'_B) \in \Pi_B^{\mathcal{O}}$ for $\mathcal{O} = (O_B \setminus I_A) \cup \mathcal{T}(A)$ as a witness for condition (A-Out) w.r.t. $C \leftrightarrow_u B$. A lemma for the "downward" direction (Lem. 4.17 below) shows how R_{AC} and $\mathscr{R}(A \otimes B)$ can be used to construct an execution fragment in $C \otimes B$ from $A \leftrightarrow_u B$, i.e. from $(s_A, l, s'_A) \in \Pi_A^{\mathcal{O}}$ for $\mathcal{O} = (O_A \setminus I_B) \cup \mathcal{T}(A)$ as a witness for condition (B-Out) w.r.t. $C \leftrightarrow_u B$.

PROOF OF PROPOSITION. Let R_{AC} be a relation witnessing $C \sqsubseteq^{\mathrm{bb}} A$ with $(s_{0,A}, s_{0,C}) \in R_{AC}$. Let $R = \{((s_A, s_B), (s_C, s_B)) \mid (s_A, s_C) \in R_{AC} \wedge (s_A, s_B) \in \mathscr{R}(A \otimes B)\}$.

Let $(s_C, s_B) \in \mathscr{R}(C \otimes B)$. Then by Lem. 4.16, there exists $s_A \in S_A$ such that $(s_A, s_B) \in \mathscr{R}(A \otimes B)$ and $(s_A, s_C) \in R_{AC}$, hence $((s_A, s_B), (s_C, s_B)) \in R$.
Let $l \in O_C \cap I_B$ and $(s_C, l, s'_C) \in \Delta_C$. By (C-I/O) for R_{AC} there exists $s'_A \in S_A$ such that $(s_A, l, s'_A) \in \Delta_A^{\mathcal{T}(A)}$. Hence, there is $\bar{s}_A, \bar{s}'_A \in S_A$ such that $(s_A, \bar{s}_A) \in \Delta_A^{\mathcal{T}(A)}$ and $(\bar{s}_A, l, \bar{s}'_A) \in \Delta_A$. By Lem. 4.3 (1), $(\bar{s}_A, s_B) \in \mathscr{R}(A \otimes B)$ and thus, by (A-Out) for $A \leftrightarrow_u B$, we have for $(\bar{s}_A, s_B)$ the existence of $s'_B \in S_B$ such that $(s_B, l, s'_B) \in \Pi_B^{\mathcal{O}_B}$ with $\mathcal{O}_B = (O_B \setminus I_A) \cup \mathcal{T}(B)$. Since $I_C = I_A$, this is a valid witness for condition (A-Out) w.r.t. $C \leftrightarrow_u B$ and $(s_C, l, s'_C) \in \Delta_C$.
Let $l \in O_B \cap I_C$ and $(s_B, l, s'_B) \in \Delta_B$. By (B-Out) for $A \leftrightarrow_u B$ we have for all $(s_A, s_B) \in \mathscr{R}(A \otimes B)$ the existence of $s'_A \in S_A$ such that $(s_A, l, s'_A) \in \Pi_A^{\mathcal{O}_A}$ where $\mathcal{O}_A = (O_A \setminus I_B) \cup \mathcal{T}(A)$. By Lem. 4.17 there exists

$$((s_C, s_B), (\bar{s}_C, s_B)) \in \Pi_{C \otimes B}^{\mathcal{O}_C} \text{ and } ((\bar{s}_C, s_B), l, (s'_C, s'_B)) \in \Delta_{C \otimes B}$$

with $\mathcal{O}_C = (O_C \setminus I_B) \cup \mathcal{T}(C)$ and $(\bar{s}_C, l, s'_C) \in \Pi_C$ (by $(s_A, l, s'_A) \in \Pi_A^{\mathcal{O}_A}$). Therefore there exists

$$\rho_C = (s_C\ l_{CB,1}\ s_{C,1}\ \ldots\ l_{CB,n}\ s_{C,n}\ l\ s'_C) \in \mathit{frag}(C)$$

with $l_{CB,i} \in \mathcal{O}_C$ and all transitions from Π_C for all $1 \leq i \leq n$. In particular $(s_{C,n}, l, s'_C) \in \Pi_C$ and thus ρ_C is a valid witness for condition (B-Out) w.r.t. $C \leftrightarrow_u B$. □

Lemma 4.16 (Upward) *Let A, B and C be PIOs. Let A, B and C, B be composable. Let $A \leftrightarrow_u B$ and let R_{AC} be a witness for $C \sqsubseteq^{\mathrm{bb}} A$. If $(s_C, s_B) \in \mathscr{R}(C \otimes B)$, then there exists $s_A \in S_A$ such that $(s_A, s_B) \in \mathscr{R}(A \otimes B)$ and $(s_A, s_C) \in R_{AC}$.*

PROOF. If $(s_C, s_B) \in \mathscr{R}(C \otimes B)$, then there exists $\rho_{CB} \in \mathit{frag}(C \otimes B)$ with

$$\rho_{CB} = ((s_{C,0}, s_{B,0})\ l_{CB,1}\ (s_{C,1}, s_{B,1})\ \ldots l_{CB,k}\ (s_C, s_B)),$$

where $l_{CB,i} \in \mathcal{L}(C \otimes B)$. By projection to C we remove all non-shared B-steps, i.e. all $((s_{C,i}, s_{B,i}), l_{CB,i}, (s_{C,i}, s_{B,i+1})$ with $l_{CB,i} \in \mathcal{L}(B) \setminus \mathcal{S}(C,B)$ and obtain

$$\rho_C^m = (s_{C,0}\ l_{C,1}\ s_{C,1}\ \ldots l_{C,m}\ s_{C,m}) \in \mathit{frag}(C),$$

with $l_{C,i} \in \mathcal{L}(C)$ and $s_{C,i} \in S_C$ for all $1 \leq i \leq m$ and $s_{C,m} = s_C$. By induction on m (see below) we construct a fragment

$$\rho_A^h = (s_{A,0}\ l_{A,1}\ s_{A,1}\ \ldots l_{A,h}\ s_{A,h}) \in \mathit{frag}(A),$$

such that for all $1 \leq j \leq h$ it holds that $l_{A,j} \in \mathcal{L}(A)$ and there exists $i \in \{0, \ldots, m\}$ such that $(s_{A,j}, s_{C,i}) \in R_{AC}$. Since ρ_A^h differs from ρ_C^m only by internal transitions not shared with B, this fragment can be used to construct

$$\rho_{AB} = ((s_{A,0}, s_{B,0})\ l_{AB,1}\ (s_{A,1}, s_{B,1})\ \ldots l_{AB,k'}\ (s_A, s_B)) \in \mathit{frag}(A \otimes B),$$

with $s_A = s_{A,h}$ using the non-shared B-steps from ρ_{CB} above, which then shows that indeed there exists $s_A \in S_A$ such that $(s_A, s_B) \in \mathscr{R}(A \otimes B)$ and $(s_A, s_C) \in R_{AC}$.

Base case ($m = 1$). Let $(s_{C,0}\ l_{C,1}\ s_{C,1}) \in \mathit{frag}(C)$. We show that there exists $s_{A,1} \in S_A$ and $s_{B,1} \in S_B$ such that $(s_{A,1}, s_{B,1}) \in \mathscr{R}(A \otimes B)$ and $(s_{A,1}, s_{C,1}) \in R_{AC}$. Since $l_{C,1} \in \mathcal{L}(C)$ we distinguish that either $l_{C,1} \in I_C \cup O_C$, or $l_{C,1} \in \mathcal{T}(C)$.
(1) Let $l_{C,1} \in I_C \cup O_C$. By (C-I/O) for $(s_{A,0}, s_{C,0}) \in R_{AC}$, there exists $s_{A,1} \in S_A$ such that $(s_{A,0}, l_{C,1}, s_{A,1}) \in \Delta_A^{\mathcal{T}(A)}$ and $(s_{A,1}, s_{C,1}) \in R_{AC}$. If $l_{C,1} \notin \mathcal{S}(C,B)$, then $l_{C,1} \notin \mathcal{S}(A,B)$ (by $I_A = I_C$ and $O_A = O_C$) and hence $(s_{A,1}, s_{B,1}) \in \mathscr{R}(A \otimes B)$ for $s_{B,1} = s_{B,0}$ by Lem. 4.3 (1). For the case $l_{C,1} \in \mathcal{S}(C,B)$ we know by the projection from the fragment ρ_{CB} above that there exists $s_{B,1} \in S_B$ such that $(s_{B,0}\ l_{C,1}\ s_{B,1}) \in \Delta_B^{\mathcal{O}_B}$ with $\mathcal{O}_B = (O_B \setminus I_A) \cup \mathcal{T}(B)$. Thus, there is $\bar{s}_B, \bar{s}'_B \in S_B$ such that $(s_{B,0}, \bar{s}_B) \in \Delta_B^{\mathcal{O}_B}$, $(\bar{s}_B, l_{C,1}, \bar{s}'_B) \in \Delta_B$ and $(\bar{s}'_B, s_{B,1}) \in \Delta_B^{\mathcal{O}_B}$. Moreover, from $(s_{A,0}, l_{C,1}, s_{A,1}) \in \Delta_A^{\mathcal{T}(A)}$ we obtain $\bar{s}_A, \bar{s}'_A \in S_A$ such that $(s_{A,0}, \bar{s}_A) \in \Delta_A^{\mathcal{T}(A)}$, $(\bar{s}_A, l_{C,1}, \bar{s}'_A) \in \Delta_A$ and $(\bar{s}'_A, s_{A,1}) \in \Delta_A^{\mathcal{T}(A)}$. By Lem. 4.3 (1), it follows that $((s_{A,0}, s_{B,0}), (\bar{s}_A, \bar{s}_B)) \in \Delta_{A \otimes B}^{\mathcal{T}(A) \cup \mathcal{O}_B}$, and thus $(\bar{s}_A, \bar{s}_B) \in \mathscr{R}(A \otimes B)$. Now $((\bar{s}_A, \bar{s}_B), l_{C,1}, (\bar{s}'_A, \bar{s}'_B)) \in \Delta_{A \otimes B}$ and, again by Lem. 4.3 (1), $((\bar{s}'_A, \bar{s}'_B), (s_{A,1}, s_{B,1})) \in \Delta_{A \otimes B}^{\mathcal{T}(A) \cup \mathcal{O}_B}$ and hence $(s_{A,1}, s_{B,1}) \in \mathscr{R}(A \otimes B)$.
(2) Let $l_{C,1} \in \mathcal{T}(C)$. By (C-Int) for $(s_{A,0}, s_{C,0}) \in R_{AC}$, there exists $s_{A,1} \in S_A$ such that $(s_{A,0}, s_{A,1}) \in \Delta_A^{\mathcal{T}(A)}$ and $(s_{A,1}, s_{C,1}) \in R_{AC}$. By Lem. 4.3 (1), $(s_{A,1}, s_{B,0}) \in \mathscr{R}(A \otimes B)$ and hence $(s_{A,1}, s_{B,1}) \in \mathscr{R}(A \otimes B)$ for $s_{B,1} = s_{B,0}$.

Induction step ($m + 1$). Let ρ_C^m and ρ_A^h be execution fragments as above such that $(s_{A,h}, s_B) \in \mathscr{R}(A \otimes B)$ and $(s_{A,h}, s_{C,m}) \in R_{AC}$ for some $m, h \in \mathbb{N}$. Assume

$$\rho_C^{m+1} = (s_{C,0}\ l_{C,1}\ s_{C,1}\ \ldots l_{C,m+1}\ s_{C,m+1}) \in \mathit{frag}(C),$$

then by induction assumption, there exists $p \geq 0$ such that $s_{A,h+p} \in S_A$ and $(s_{A,h+p}, s_B) \in \mathscr{R}(A \otimes B)$ and $(s_{A,h+p}, s_{C,m+1}) \in R_{AC}$. $\square$

Lemma 4.17 (Downward) *Let A, B and C be PIOs. Let R_{AC} be a witness for $C \sqsubseteq^{\mathrm{bb}} A$. Let $(s_A, s_C) \in R_{AC}$. Let $l \in O_B \cap I_C$ and $s'_B \in S_B$ such that $(s_B, l, s'_B) \in \Delta_B$. If $(s_A, s_B) \in \mathscr{R}(A \otimes B)$ and there exists $s'_A \in S_A$ such that $(s_A, l, s'_A) \in \Pi_A^{\mathcal{O}_A}$, where $\mathcal{O}_A = (O_A \setminus I_B) \cup \mathcal{T}(A)$, then there exists $\bar{s}_C, s'_C \in S_C$ such that $((s_C, s_B), (\bar{s}_C, s_B)) \in \Pi_{C \otimes B}^{\mathcal{O}_C}$ and $((\bar{s}_C, s_B), l, (s'_C, s'_B)) \in \Delta_{C \otimes B}$, where $\mathcal{O}_C = (O_C \setminus I_B) \cup \mathcal{T}(C)$.*

PROOF. By $(s_A, l, s'_A) \in \Pi_A^{\mathcal{O}_A}$ there exists $\bar{s}_A, \bar{s}'_A \in S_A$ such that $(s_A, \bar{s}_A) \in \Pi_A^{\mathcal{O}_A}$, $(\bar{s}_A, l, \bar{s}'_A) \in \Pi_A$ and $(\bar{s}'_A, s'_A) \in \Pi_A^{\mathcal{O}_A}$. Hence there exist $l_{A,i} \in \mathcal{O}_A$ and $s_{A,i} \in S_A$ for $1 \leq i < n$ such that $(s_A, l_{A,1}, s_{A,1}) \in \Pi_A$, $(s_{A,i}, l_{A,i+1}, s_{A,i+1}) \in \Pi_A$, …, $(s_{A,n}, l, \bar{s}'_A) \in \Pi_A$ with $\bar{s}_A = s_{A,n}$. Below we show by induction on n that there exists $\bar{s}_C \in S_C$ such that $(s_C, \bar{s}_C) \in \Pi_C^{\mathcal{O}_C}$ with $\mathcal{O}_C = (O_C \setminus I_B) \cup \mathcal{T}(C)$ and $(\bar{s}_A, \bar{s}_C) \in R_{AC}$. Then, by Lem. 4.3 for $(s_C, \bar{s}_C) \in \Pi_C^{\mathcal{O}_C}$ we have $((s_C, s_B), (\bar{s}_C, s_B)) \in \Pi_{C \otimes B}^{\mathcal{O}_C}$. Now,

since $(\bar{s}_A, \bar{s}_C) \in R_{AC}$ and $(\bar{s}_A, l, \bar{s}'_A) \in \Pi_A$ there exists $\bar{s}'_C \in S_C$ such that $(\bar{s}'_A, \bar{s}'_C) \in R_{AC}$ and $(\bar{s}_C, l, \bar{s}'_C) \in \Pi_C^{\mathcal{T}(C)}$ by (A-I/O) for R_{AC}. Thus by Lem. 4.3 for $(\bar{s}_C, l, \bar{s}'_C) \in \Pi_C^{\mathcal{T}(C)}$ and $(s_B, l, s'_B) \in \Delta_B$ it follows that $((\bar{s}_C, s_B), l, (\bar{s}'_C, s'_B)) \in \Pi_{C \otimes B}$.

Base case ($n = 1$). Let $(s_A, l_{A,1}, s_{A,1}) \in \Pi_A$ with $l_{A,1} \in \mathcal{O}_A$. There are two cases. If $l_{A,1} \in O_A \setminus I_B$, then by (A-I/O) for R_{AC} there is $s_{C,1} \in S_C$ such that $(s_C, l_{A,1}, s_{C,1}) \in \Pi_C^{\mathcal{T}(C)}$ and $(s_{A,1}, s_{C,1}) \in R_{AC}$. Since $O_A = O_C$ it follows that $(s_C, s_{C,1}) \in \Pi_C^{\mathcal{O}_C}$.
If $l_{A,1} \in \mathcal{T}(A)$, then by (A-Int) for R_{AC} there is $s_{C,1} \in S_C$ such that $(s_C, s_{C,1}) \in \Pi_C^{\mathcal{T}(C)}$ and $(s_{A,1}, s_{C,1}) \in R_{AC}$. By $\mathcal{T}(C) \subseteq \mathcal{O}_C$ it follows that $(s_C, s_{C,1}) \in \Pi_C^{\mathcal{O}_C}$.

Induction step ($n + 1$). Let $s_{A,n} \in S_A$ and $s_{c,m} \in S_C$ such that $(s_{A,n}, s_{C,m}) \in R_{AC}$ and $(s_C, s_{C,m}) \in \Pi_C^{\mathcal{O}_C}$ for some $n, m \in \mathbb{N}$. Assume $l_{A,n+1} \in \mathcal{O}_A$ and $s_{A,n+1} \in S_A$ such that $(s_{A,n}, l_{A,n+1}, s_{A,n+1}) \in \Pi_A$. Then by induction assumption there exists $s_{c,m+1} \in S_C$ such that $(s_{A,n+1}, s_{C,m+1}) \in R_{AC}$ and $(s_C, s_{C,m+1}) \in \Pi_C^{\mathcal{O}_C}$. □

Example 4.18 (Ultra-weak preservation) *Since BufIn* $\otimes$ *BufOut* $\sqsubseteq^{\mathrm{bb}}$ *Buf (Ex. 4.6) and Prod* $\leftrightarrow_u$ *Buf (Ex. 4.14), Prop. 4.15 allows to conclude ultra-weak output compatibility between Prod and BufIn*$\otimes$*BufOut. Note that condition (A-I/O) of* $\sqsubseteq^{\mathrm{bb}}$ *forces the buf-transition of BufIn* $\otimes$ *BufOut to be persistent. Otherwise the state* $((b_1), (bi_1, bo_0))$ *would not be in relation since the input-transition put would not be reachable via persistent transitions in* $\Pi_{\mathit{BufIn} \otimes \mathit{BufOut}}$*. In contrast, the output-transitions get are not directly forced to be persistent. However, by Prod* $\leftrightarrow_u$ *Buf it is required that* $\{(b_1, \mathit{get}, b_0), (b_2, \mathit{get}, b_1)\} \subseteq \Pi_{\mathit{Buf}}$ *and now, condition (A-I/O) of* $\sqsubseteq^{\mathrm{bb}}$ *ensures again that these transitions indeed exist as persistent transitions in* $\Pi_{\mathit{BufIn} \otimes \mathit{BufOut}}$ *of the concrete PIO.*

Preservation of weak output compatibility holds by Prop. 4.7 together with the correspondence of weak output compatibility for PIOs and weak compatibility for MIOs, and the preservation theorem in [**BMSH10**, Thm. 2].

Proposition 4.19 (Weak preservation) *Let* A*,* B *and* C *be PIOs such that* A*,* B *and* C*,* B *are composable. If* $A \leftrightarrow_w B$ *and* $C \sqsubseteq^{\mathrm{bb}} A$*, then* $C \leftrightarrow_w B$*.* □

2. Asynchronous Communication

One of the major aims of this study is to develop a formal model which takes asynchronous communication in the form of FIFO buffered message exchange into account. The theory developed so far works for PIOs which synchronise by a rendezvous mechanism. Within this section we extend our theory by an encoding of FIFO buffers (queues) for the modelling of asynchronous buffered communication and a definition of asynchronous compatibility which is based on weak and ultra-weak compatibility respectively. After that we show first, that blackbox refinement can be transfered to PIOs with FIFO buffered communication and second, that blackbox refinement also preserves asynchronous compatibility. We start with some preliminaries concerning basic definitions and notations from the domain of I/O-transition systems (cf. Chap. 2, Sect. 2) and their transfer to the domain of PIOs with asynchronous communication.

A partitioned I/O-labelling is an I/O-labelling $((I_1, O_1), \ldots, (I_n, O_n), T)$, where inputs and outputs are given by a partition $\{(I_1, O_1), \ldots, (I_n, O_n)\}$. Composability of partitioned IOLs requires, besides the general composability conditions, a unique assignment of shared labels to the elements of the partition. A *partitioned PIO* is a PIO with partitioned I/O-labelling.

Partitioned I/O-labellings are used to distinguish the queues of buffered transition systems. Intuitively an element of a partitioned I/O-labelling corresponds to a set of labels as given by the port of a port-based component. Then for the required interface of the port there will be an output queue which acts as a buffer for the messages sent into one direction. For binary connectors the resulting communication medium results in two output queues, one for each direction as illustrated in Fig. 4.6 showing two systems A and B,

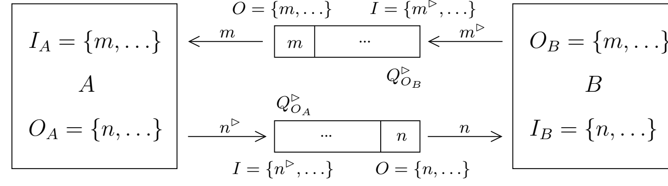


FIGURE 4.6. Transition systems with output queues

each with one output queue $Q^\triangleright_{O_A}$ and $Q^\triangleright_{O_B}$ defined with respect to the output alphabet of the respective PIO.

In general, there will be several output queues. The queues are encoded using PIOs. In order to synchronise such a system of transition systems and queues appropriately we need to apply a relabelling such that output messages are synchronised with the particular queue transition system. The outputs of a queue then replace the original outputs. A corresponding relabelling is indicated in Fig. 4.6. By this means, message exchange is buffered and PIOs do not synchronise directly anymore. Note that the indicated output relabelling is transparent for the communication partner.

Let $L = ((I_1, O_1), \dots, (I_n, O_n), T)$ be a partitioned IOL. An *asynchronous output labelling* with respect to L is given by $O^\triangleright = (O^\triangleright_1, \dots, O^\triangleright_k)$ for some $k \leq n$ where $O^\triangleright_i$ is defined by $l^\triangleright \in O^\triangleright_i$ if $l \in O_i$, for all $1 \leq i \leq k$. An asynchronous output labelling for a PIO A is given by an asynchronous output labelling $O^\triangleright_A$ with respect to L_A.

Definition 4.20 (αPIO) *A* PIO with asynchronous output labelling *(αPIO for short) is a pair $(A, O^\triangleright)$ such that A is a partitioned PIO and $O^\triangleright$ an asynchronous output labelling of A. If $|O^\triangleright| = |O_A|$ we simply write* A is a αPIO .

The *relabelling of A for asynchronous communication with respect to $O^\triangleright$* is given by a relabelling β from L_A to L'_A defined by a function $f : \bigcup L_A \to \bigcup L'_A$ such that $f(l) = l^\triangleright$ if $l^\triangleright \in O^\triangleright$, and $f(l) = l$ else. Relabelling of a PIO for asynchronous communication should not be applied more than once for the same set of labels. Therefore, labels given by an asynchronous output labelling must not yet exist in the set of labels of the underlying PIO. We require for any αPIO $(A, O^\triangleright)$ that $\bigcup L_A \cap \bigcup O^\triangleright = \emptyset$.

Concerning the composability of αPIOs we need to ensure that the asynchronous output labels do not invalidate composability requirements of PIOs. Two *αPIOs $(A, O^\triangleright_A)$ and $(B, O^\triangleright_B)$ are composable* if A and B are composable and $\bigcup L_A \cap \bigcup O^\triangleright_B = \emptyset$ and $\bigcup L_B \cap \bigcup O^\triangleright_A = \emptyset$.

2.1. FIFO Encoding and Buffered PIOs. We define an encoding of FIFO queues using PIOs which relies on the definition of output queues in the context of I/O-transition systems in Chap. 2. Since IOTSs are a formal representation of component implementations (behaviours), this highlights that queues are indeed not supposed to be specified to allow for several different implementations of a queue. Instead we will use the same implementation-related formalism of FIFO queues on both levels, on the level of specifications (PIOs) and on the level of implementations (IOTSs).

Definition 4.21 (FIFO Encoding) *The* PIO encoding of a FIFO queue with respect to a set M, *written $Q^\triangleright_M$, is given by $Q^\triangleright_M = (L, S, s_0, \Delta, \Pi)$, where (L, S, s_0, Δ) is a queue IOTS over M and $\Pi = \Delta$.*

An asynchronous output labelling specifies which messages of the given transition system should be exchanged using FIFO buffers. With Def. 4.21 it is straightforward to define a corresponding notion of buffered PIOs.

Definition 4.22 (Buffered PIO) *The* buffered PIO *for an αPIO $(A, (O_1, \ldots, O_k))$ for some $k \leq |O_A|$ is defined by*

$$\Omega(A, (O_1, \ldots, O_k)) = (A\beta \otimes Q^{\triangleright}_{O_1} \otimes \cdots \otimes Q^{\triangleright}_{O_k}),$$

where β is a relabelling of A for asynchronous communication w.r.t. $(O_1, \ldots, O_k)$. It is called completely buffered*, written $\Omega(A)$, if $k = |O_A|$. We abbreviate the product of output queues $Q^{\triangleright}_{O_1} \otimes \cdots \otimes Q^{\triangleright}_{O_k}$ by $Q^{\triangleright}_A$.*

Next we ensure that buffered PIOs can be used just like ordinary PIOs. In particular, their composition is associative and commutative using the product operator for PIOs. Also note that $\Omega(A, \emptyset) = A$.

Lemma 4.23 *Let $(A, O^{\triangleright}_A)$ and $(B, O^{\triangleright}_B)$ be αPIOs. Then the following holds:*

(1) $\Omega(A, O^{\triangleright}_A)$ is a PIO with $L_{\Omega(A,O^{\triangleright}_A)} = (I_A, O_A, T_A \cup \bigcup O^{\triangleright}_A)$.

(2) If $(A, O^{\triangleright}_A)$ and $(B, O^{\triangleright}_B)$ are composable then $\Omega(A, O^{\triangleright}_A)$ and $\Omega(B, O^{\triangleright}_B)$ are composable.

PROOF. (1) The output queues for $(A, O^{\triangleright}_A)$ are composable, since $O^{\triangleright}_A$ is a partition and, by definition, queue inputs are disjoint from queue outputs and the sets of internal labels are empty. Thus $Q^{\triangleright}_A$ is defined. Let $\Omega(A, O^{\triangleright}_A) = (A\beta \otimes Q^{\triangleright}_A)$; $A\beta$ and $Q^{\triangleright}_A$ are composable since $(A, O^{\triangleright}_A)$ is an αPIO, that is, $A\beta$ is a valid PIO and $I_{A\beta} \cap I_{Q^{\triangleright}_A} = \emptyset$, $O_{A\beta} \cap O_{Q^{\triangleright}_A} = \emptyset$, $T_{A\beta} \cap \bigcup Q^{\triangleright}_{sA} = \emptyset$ and $T_{Q^{\triangleright}_A} \cap \bigcup A\beta = \emptyset$ by definition. Hence $\Omega(A, O^{\triangleright}_A)$ is a PIO. Now, by definition of IOL products, $I_{\Omega(A,O^{\triangleright}_A)} = (I_{A\beta} \cup \bigcup I_{Q^{\triangleright}_A}) \setminus \mathcal{S}(A\beta, Q^{\triangleright}_A) = I_A$, $O_{\Omega(A,O^{\triangleright}_A)} = (O_{A\beta} \cup \bigcup O_{Q^{\triangleright}_A}) \setminus \mathcal{S}(A\beta, Q^{\triangleright}_A) = O_{Q^{\triangleright}_A} = O_A$ and $T_{\Omega(A,O^{\triangleright}_A)} = T_{A\beta} \cup \bigcup T_{Q^{\triangleright}_A} \cup \mathcal{S}(A\beta, Q^{\triangleright}_A) = T_A \cup \bigcup O^{\triangleright}_A$.

(2) Using (1) we need to proof that $I_A \cap I_B = \emptyset$, $O_A \cap O_B = \emptyset$, $(T_A \cup \bigcup O^{\triangleright}_A) \cap \bigcup O^{\triangleright}_B = \emptyset$ and $(T_B \cup \bigcup O^{\triangleright}_B) \cap \bigcup O^{\triangleright}_A = \emptyset$, which is a consequence of composability of $(A, O^{\triangleright}_A)$ and $(B, O^{\triangleright}_B)$. Moreover, since $I_{\Omega(A,O^{\triangleright}_A)} = I_A$ and $O_{\Omega(A,O^{\triangleright}_A)} = O_A$, we have that shared labels will still be uniquely assigned to elements of the partitions and hence also the requirement for the composability w.r.t partitions is preserved. $\Box$

Example 4.24 (Buffered PIO) *Buffered PIOs are used to model buffered communication between processes modelled by finite-state control automata, in our case by PIOs. With the buffered producer example in Ex. 4.2 we considered a producer PIO composed with an output buffer, which encodes its capacity explicitely by a corresponding number of input transitions* put*, followed by an output transition* get*. Instead of this explicit modelling of the buffer's capacity, we could use buffered PIOs to abstract from capacity issues and focus solely on the control part, which is in particular useful for the modelling of a distributed system. In this case, the buffer between producer and consumer not only models storage but is also used to decouple and control the message exchange.*

For instance, the usual modelling of a producer/consumer-architecture requires the consumer to wait with getting items from the storage as long as there are no items available, i.e. as long as the buffer is empty. Such a requirement could be modelled with a buffer process BufCtr *as depicted in Fig. 4.7.*

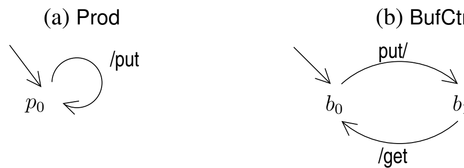


FIGURE 4.7. PIOs for producer and buffer controller

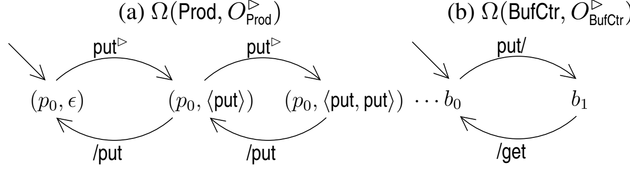


FIGURE 4.8. Buffered PIOs for the transition systems in Fig. 4.7

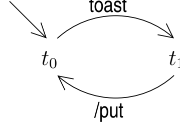


FIGURE 4.9. Refinement of the original producer

Then, asynchronous (buffered) communication between Prod *and* BufCtr *is introduced by a corresponding definition of buffered PIOs.* $\Omega(\mathsf{Prod}, O^{\rhd}_{\mathsf{Prod}})$ *with* $O^{\rhd}_{\mathit{Prod}} = \{(\emptyset, \{\mathit{put}\})\}$ *results in a PIO with an output queue for messages labelled* put*. An excerpt of the corresponding (infinite-state) transition system is depicted in Fig. 4.8. For the buffer control* BufCtr *we leave the output* get *open, hence define* $O^{\rhd}_{\mathit{BufCtr}} = \{(\{\mathit{put}\}, \emptyset)\}$*. Thus,* $\Omega(\mathsf{BufCtr}, O^{\rhd}_{\mathsf{BufCtr}})$ *is still a PIO without queue which is obtained from* BufCtr *by a relabelling of* put *for asynchronous communication (cf. Fig. 4.8). The composition* $\Omega(\mathsf{Prod}, O^{\rhd}_{\mathsf{Prod}}) \otimes \Omega(\mathsf{BufCtr}, O^{\rhd}_{\mathsf{BufCtr}})$ *then results in an infinite-state system, which repeatedly (and internally) produces items, stores them in an internal FIFO buffer and then externally provides them for consumption via the open output transition* get*.*

2.2. Refinement and Asynchronous Compatibility. Our component model explicitly distinguishes components and connectors. The components' behaviour is specified by automata showing a temporal ordering of receive, send and local actions. Connectors in contrast do not specify any application specific temporal ordering of message exchange but act as a communication medium which determines the timing of the message exchange between components linked with this connector. Therefore, connectors are not user-defined in the sense components are and hence our notion of refinement must not apply to connectors. Instead we focus on the refinement of the components' behaviour aiming at a notion of compositionality which is independent of the connectors used on the abstract system level. For the case of synchronous communication this is backed by Prop. 4.12 already. For the case of asynchronous communication we rely on the following proposition.

Theorem 4.25 (Refinement transfer) *Let* $(A, O^{\rhd})$ *be an* α*PIO and let* C *be a partitioned PIO. If* $C \sqsubseteq^{\mathrm{bb}} A$*, then* $\Omega(C, O^{\rhd}) \sqsubseteq^{\mathrm{bb}} \Omega(A, O^{\rhd})$*.*

PROOF. Let β be the relabelling for asynchronous communication w.r.t. $O^{\rhd}$. By $C \sqsubseteq^{\mathrm{bb}} A$ we have $C\beta \sqsubseteq^{\mathrm{bb}} A\beta$ (similar to Lem. 2.16 (2)). Denote the output queues for $O^{\rhd}$ by $Q^{\rhd}$ then it follows by Thm. 4.10 that $C\beta \otimes Q^{\rhd} \sqsubseteq^{\mathrm{bb}} C\beta \otimes Q^{\rhd}$. □

Example 4.26 (Refinement transfer) *Suppose the producer of Ex. 4.24 creates a slice of toasted bread which should be passed to the consumer afterwards. In a corresponding process as depicted in Fig. 4.9 toast production is an internal step preceeding the storage of the produced item via the (yet) open output transition* put*. If the internal transition is persistent, i.e.* $(t_0, \mathit{toast}, t_1) \in \Pi_{\mathit{Toast}}$ *we have* Toast $\sqsubseteq^{\mathrm{bb}}$ Prod*, witnessed by the relation* $\{(t_0, p_0), (t_1, p_0)\} \subseteq S_{\mathit{Toast}} \times S_{\mathit{Prod}}$*. Therefore Thm. 4.25 is applicable and it follows that* $\Omega(\mathit{Toast}, O^{\rhd}_{\mathit{Prod}}) \sqsubseteq^{\mathrm{bb}} \Omega(\mathsf{Prod}, O^{\rhd}_{\mathsf{Prod}})$*.*

Note that $\Omega(\mathsf{Prod}, O^{\triangleright}_{\mathsf{Prod}})$ *is an infinite-state system and hence verification of its refinement is in general undecidable. In contrast,* *Toast* $\sqsubseteq^{\mathrm{bb}}$ *Prod could be verified using finite-state verification techniques.*

Based on Thm. 4.25 we can claim compositionality for the case of buffered PIOs similar to the case of PIOs with rendezvous communication. Though note that there is an important difference: Here we rely on the refinement of the PIOs underlying the particular buffered PIOs and not on a refinement between the buffered PIOs in turn.

Corollary 4.27 (Precongruence transfer) *Let* $(A, O^{\triangleright}_A)$ *and* $(B, O^{\triangleright}_B)$ *be composable* α*PIOs and let* C *be a partitioned PIO. Let* C *and* B *be composable. If* $C \sqsubseteq^{\mathrm{bb}} A$*, then* $\Omega(C, O^{\triangleright}_A) \otimes \Omega(B, O^{\triangleright}_B) \sqsubseteq^{\mathrm{bb}} \Omega(A, O^{\triangleright}_A) \otimes \Omega(B, O^{\triangleright}_B)$

PROOF. The claim follows by Thm. 4.25 and Thm. 4.10. □

Example 4.28 (Precongruence transfer) *Applying the corollary to Ex. 4.26 allows the buffered producer to be replaced by a buffered toast producer in the composition with the buffer controller. More formally, since* *Toast* $\sqsubseteq^{\mathrm{bb}}$ *Prod (cf. Ex. 4.26) we have, by Cor. 4.27,*

$$\Omega(\mathit{Toast}, O^{\triangleright}_{\mathit{Prod}}) \otimes \Omega(\mathit{BufCtr}, O^{\triangleright}_{\mathit{BufCtr}}) \sqsubseteq^{\mathrm{bb}} \Omega(\mathit{Prod}, O^{\triangleright}_{\mathit{Prod}}) \otimes \Omega(\mathit{BufCtr}, O^{\triangleright}_{\mathit{BufCtr}}).$$

Corollary 4.29 (Compositionality transfer) *Let* $(A, O^{\triangleright}_A)$ *and* $(B, O^{\triangleright}_B)$ *be composable* α*PIOs and let* C*,* D *be composable partitioned PIOs. If* $C \sqsubseteq^{\mathrm{bb}} A$ *and* $D \sqsubseteq^{\mathrm{bb}} B$*, then* $\Omega(C, O^{\triangleright}_A) \otimes \Omega(D, O^{\triangleright}_B) \sqsubseteq^{\mathrm{bb}} \Omega(A, O^{\triangleright}_A) \otimes \Omega(B, O^{\triangleright}_B)$

PROOF. The proof is analogous to the synchronous case (cf. Prop. 4.12). □

Another important aim of our study is to understand which basic compatibility requirements may apply to the case of buffered message exchange. This corresponds to the question which kind of architectural incompatibilities should be avoided when using asynchronous connectors (cf. Chap. 2, Sect. 1.3). Considering our notion of output compatibility as defined above, we aim at guarantees for sending messages since the reception of a message already implies that some sender was successful in omitting the particular message which is not the case for sending. In the context of buffered message exchange this transfers to some guarantee that, once a message has been put into the output queue of a PIO, the particular communication partner is indeed able to dequeue the corresponding message later on. The transition systems in Fig. 4.10 and Fig. 4.11 illustrate such a requirement more concretely. In Fig. 4.10 both systems are indeed able to dequeue the message sent by its communication partner, either before or after sending messages in turn. For the transition systems in Fig. 4.11, however, this is not the case. Here the fundamental difference between a rendezvous mechanism and buffered message exchange becomes evident. If both systems send messages at the "same" time, the asynchronous system deadlocks due to missing receive capabilities after having sent the message n and m respectively.

For the formal definition of a corresponding notion of asynchronous compatibility we rely on the notions of weak, ultra-weak compatibility defined above which is due to the encoding of FIFO buffers along transition systems for the messages sent (output queues) in contrast to an encoding for messages received (input queues).

Definition 4.30 (Asynchronous output compatibility) *Let* $(A, O^{\triangleright}_A)$ *and* $(B, O^{\triangleright}_B)$ *be composable* α*PIOs.* $(A, O^{\triangleright}_A)$ *and* $(B, O^{\triangleright}_B)$ *are* ultra-weakly asynchronous output compatible*, written* $(A, O^{\triangleright}_A) \leftrightarrow_a (B, O^{\triangleright}_B)$*, if* $\Omega(A, O^{\triangleright}_A) \leftrightarrow_u \Omega(B, O^{\triangleright}_B)$*; they are* weakly asynchronous output compatible *if* $\Omega(A, O^{\triangleright}_A) \leftrightarrow_w \Omega(B, O^{\triangleright}_B)$.

If not explicitely mentioned, we consider $(A, O^{\triangleright}_A) \leftrightarrow_a (B, O^{\triangleright}_B)$ and simply talk of asynchronously output compatible αPIOs.

Example 4.31 (Asynchronous output compatibility) *Consider the transition systems* A *and* B *in Fig. 4.10 and assume* $\mathcal{S}(A, B) = \{n, m\}$*, then* A *and* B *are not synchronously*

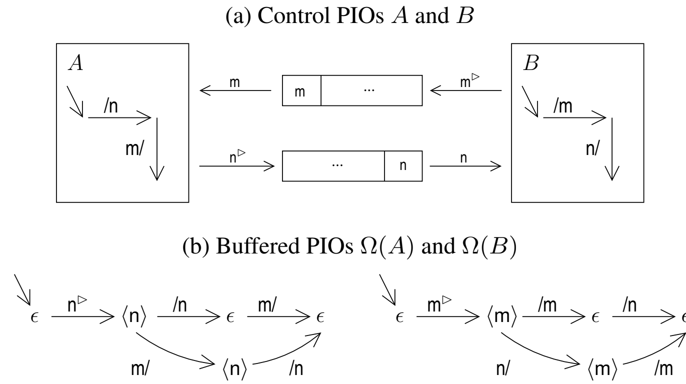


FIGURE 4.10. Simultaneous sending of messages: asynchronously compatible

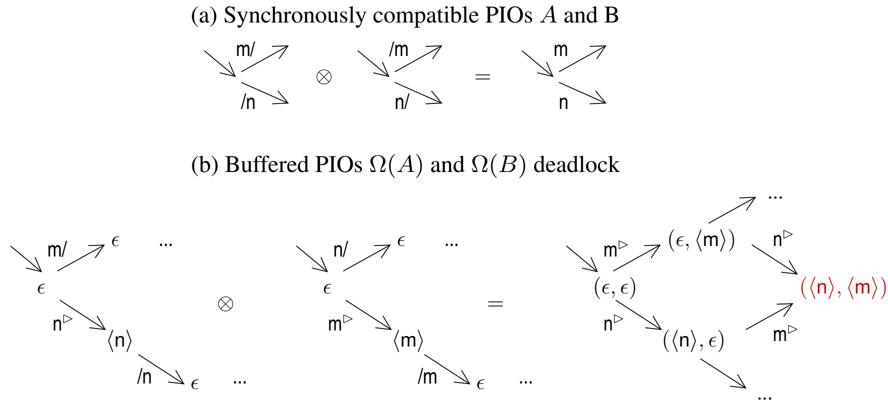


FIGURE 4.11. Simultaneous sending and mixed-choice: not asynchronously compatible

ultra-weak output compatible, since neither does A provide an input m before sending n, nor does B for the case of n the other way around. Note that $\mathcal{S}(A, B) = \{n, m\}$ is the crucial prerequisite for this example. However, $(A, O^{\triangleright}_A)$ and $(B, O^{\triangleright}_B)$ are asynchronously output compatible if $O^{\triangleright}_A = \{n\}$ and $O^{\triangleright}_B = \{m\}$, i.e. if the respective outputs are communicated with asynchronous timing. More formally, by definition we have $(A, O^{\triangleright}_A) \leftrightarrow_a (B, O^{\triangleright}_B)$, if $\Omega(A, O^{\triangleright}_A) \leftrightarrow_u \Omega(B, O^{\triangleright}_B)$. Thus, under the assumption $L_A = O^{\triangleright}_A$ and $L_B = O^{\triangleright}_B$, we need to check ultra-weak output compatibility for the buffered PIOs $\Omega(A)$ and $\Omega(B)$ in Fig. 4.10, which is easily verified either by manual inspection or, in this case, even by finite-state verification techniques.

In contrast, the transition systems A and B in Fig. 4.11 are synchronously ultra-weak output compatible but not asynchronously output compatible. The latter is due to the buffering of output messages, which introduces the possibility of deadlock, whenever a mixed-choice state (a state with both an input and an output transition leaving) is left by both communication partner using shared output transitions at the same time.

Asynchronous compatibility is preserved by blackbox refinement of the PIOs underlying the given buffered PIOs. This preservation result is particularly useful from the verification point of view due to the potential infinite state space of buffered PIOs. Once asynchronous compatibility is verified, modular refinement checks for the PIOs underlying the given buffered PIOs are sufficient to conclude asynchronous compatibility of the more concrete composition.

Theorem 4.32 (Preservation) *Let $(A, O^{\triangleright}_A)$ and $(B, O^{\triangleright}_B)$ be composable αPIOs and let C be a partitioned PIO. Let C and B be composable. If $C \sqsubseteq^{\mathrm{bb}} A$ and $(A, O^{\triangleright}_A) \leftrightarrow_a (B, O^{\triangleright}_B)$, then $(C, O^{\triangleright}_A) \leftrightarrow_a (B, O^{\triangleright}_B)$.*

PROOF. By Thm. 4.25 we have $\Omega(C, O^{\triangleright}_A) \sqsubseteq^{\mathrm{bb}} \Omega(A, O^{\triangleright}_A)$. By definition of asynchronous output compatibility we have $\Omega(A, O^{\triangleright}_A) \leftrightarrow_u \Omega(B, O^{\triangleright}_B)$, hence Prop. 4.15 is applicable and we obtain $\Omega(C, O^{\triangleright}_A) \leftrightarrow_u \Omega(B, O^{\triangleright}_B)$, which means that $(C, O^{\triangleright}_A) \leftrightarrow_a (B, O^{\triangleright}_B)$, again by definition of asynchronous compatibility. □

Example 4.33 (Preservation) *The αPIOs $(\mathsf{Prod}, O^{\triangleright}_{\mathsf{Prod}})$ and $(\mathsf{BufCtr}, O^{\triangleright}_{\mathsf{BufCtr}})$ are asynchronously output compatible due to ultra-weak output compatibility of $\Omega(\mathsf{Prod}, O^{\triangleright}_{\mathsf{Prod}})$ and $\Omega(\mathsf{BufCtr}, O^{\triangleright}_{\mathsf{BufCtr}})$. Consider the buffered PIOs in Fig. 4.8. The buffered producer may always eventually reach a state which outputs $\mathsf{put}?$ with one step, since all $\mathsf{put}!$ transitions are internal transitions. $\Omega(\mathsf{BufCtr}, O^{\triangleright}_{\mathsf{BufCtr}})$ is then required, by the conditions of ultra-weak output compatibility, to reach a global state with a matching input transition (labelled $\mathsf{put}?$) after at most internal and non-shared output transitions. The buffered PIO $\Omega(\mathsf{BufCtr}, O^{\triangleright}_{\mathsf{BufCtr}})$ in Fig. 4.8 obviously meets these requirements and therefore, we have indeed $(\mathsf{Prod}, O^{\triangleright}_{\mathsf{Prod}}) \leftrightarrow_a (\mathsf{BufCtr}, O^{\triangleright}_{\mathsf{BufCtr}})$.*
If the producer is replaced by a more concrete process such as Toast (cf. Ex. 4.26), then asynchronous output compatibility is preserved if $\mathsf{Toast} \sqsubseteq^{\mathrm{bb}} \mathsf{Prod}$. Put differently, once we know about asynchronous output compatibility, it suffices to check finite-state PIOs in order to ensure its preservation. In the example Thm. 4.32 is applicable due to $\mathsf{Toast} \sqsubseteq^{\mathrm{bb}} \mathsf{Prod}$ and thus it follows from $(\mathsf{Prod}, O^{\triangleright}_{\mathsf{Prod}}) \leftrightarrow_a (\mathsf{BufCtr}, O^{\triangleright}_{\mathsf{BufCtr}})$ that $(\mathsf{Toast}, O^{\triangleright}_{\mathsf{Prod}}) \leftrightarrow_a (\mathsf{BufCtr}, O^{\triangleright}_{\mathsf{BufCtr}})$.

3. Compatibility and N-ary Composition

In this section we show how to transfer the binary notion of output compatibility to a notion of output compatibility for n-ary compositions called *communication-safety* and discuss possibilities to derive this global property from pairwise or incremental compatibility analysis respectively. We consider only the case of synchronous communication, since buffered PIOs are PIOs (cf. Prop. 4.23) and thus any definition and result holds analogously in the context of asynchronous communication[1]. Therefore we will use the notion of communication-safety for compositions of PIOs as well as for compositions of buffered PIOs. In the latter case we assume a definition analogous to Def. 4.34 below using αPIOs $(A, O^{\triangleright}_A)$ instead of A.

3.1. Communication-Safety. The notion of communication-safety is a direct transfer of the binary definition of output compatibility to an n-ary composition of PIOs. If a global state in $\mathscr{R}(\otimes_i A_i)$ is safe, then there is for any globally reachable local output action a reachable local input such that the output may be synchronised within the global product.

Definition 4.34 (Communication-safety) *Let $(A_1, \dots, A_n)$ be pairwise composable PIOs with $A_i = ((I_i, O_i, T_i), S_i, s_{0,i}, \Delta_i, \Pi_i)$. A state $(s_1, \dots, s_n) \in \mathscr{R}(\bigotimes_i A_i)$ is* communication-safe (comm-safe) *for A_k with $k \in \{1, \dots, n\}$, if the following holds:*

$$\forall l \in O_k \cap (\textstyle\bigcup_i I_i) \;.\; \exists s'_k \in S_k \;.\; (s_k, l, s'_k) \in \Delta_k \implies$$
$$(\exists j \in \{1, \dots, n\} \setminus \{k\} \;.\; \exists s'_j \in S_j \;.\; (s_j, l, s'_j) \in \Pi_j^{\mathcal{O}_j}),$$

where $\mathcal{O}_j \subseteq (O_j \setminus I_k) \cup \mathcal{T}(A_j)$. The IOTS A_k is comm-safe in $(A_1, \dots, A_n)$, if all $(s_1, \dots, s_n) \in \mathscr{R}(\bigotimes_i A_i)$ are comm-safe for A_k. $(A_1, \dots, A_n)$ is comm-safe, if A_k is comm-safe in $(A_1, \dots, A_n)$ for all $1 \leq k \leq n$. Ultra-weak comm-safety *is defined using comm-safety with $\mathcal{O} = (O_j \setminus I_k) \cup \mathcal{T}(A_j)$,* weak comm-safety *with $\mathcal{O}_j = \mathcal{T}(A_j)$ and* strong comm-safety *with $\mathcal{O}_j = \emptyset$.*

---

[1]Neither the definitions nor the results of the given section assume finite transition systems.

Blackbox refinement preserves weak and ultra-weak comm-safety. Strong comm-safety, however, is not preserved due to the possibility of adding and removing internal transitions in refined PIOs.

Proposition 4.35 (Preservation) *Let $A_1, \ldots, A_n$ be pairwise composable PIOs and let $k \in \{1, \ldots, n\}$. Let C be a PIO such that C is composable with A_i for all $i \neq k$. If $C \sqsubseteq^{\mathrm{bb}} A_k$ and $(A_1, \ldots, A_n)$ is weakly (ultra-weakly) comm-safe, then $(A_1, \ldots, A_{k-1}, C, A_{k+1}, \ldots A_n)$ is weakly (ultra-weakly) comm-safe.*

The proof relies on the following lemma.

Lemma 4.36 *Let $(A_1, \ldots, A_n)$ be pairwise composable PIOs. If $(A_1, \ldots, A_n)$ is comm-safe, then $A_k \leftrightarrow (A_1 \otimes \cdots \otimes A_{k-1} \otimes A_{k+1} \otimes \cdots \otimes A_n)$ for all $k \in \{1, \ldots, n\}$.*

PROOF OF LEMMA. Let $A_R = (A_1 \otimes \cdots \otimes A_{k-1} \otimes A_{k+1} \otimes \cdots \otimes A_n)$. By comm-safety of $(A_1, \ldots, A_n)$ it holds that A_k is comm-safe in $(A_1, \ldots, A_n)$, thus all reachable states of $\otimes_i(A_i)$ are comm-safe for A_k. Therefore, the reachable states of $A_k \otimes A_R$ are comm-safe for A_k and we have for all $(s_k, s_R) \in \mathscr{R}(A_k \otimes A_R)$ that

$$\forall l \in O_{A_k} \cap I_{A_R} \,.\, \exists s'_k \in S_{A_k} \,.\, (s_k, l, s'_k) \in \Delta_{A_k} \implies$$
$$\exists s'_R \in S_{A_R} \,.\, (s_R, l, s'_R) \in \Pi^{\mathcal{O}_R}_{A_R},$$

with $\mathcal{O}_R \subseteq (O_{A_R} \setminus I_{A_k}) \cup \mathcal{T}(A_R)$. Thus the compatibility condition (A-Out) holds for A_k and A_R. The proof for condition (B-Out) is analogous in the direction from A_R to A_k. □

PROOF OF PROPOSITION. Let $A_R = (A_1 \otimes \cdots \otimes A_{k-1} \otimes A_{k+1} \otimes \cdots \otimes A_n)$. By Lem. 4.36 it follows that $A_k \leftrightarrow_w A_R$ and $A_k \leftrightarrow_u A_R$. Thus by $C \sqsubseteq^{\mathrm{bb}} A_k$, Prop. 4.19 and Prop. 4.15 we have $C \leftrightarrow_w A_R$ and $C \leftrightarrow_u A_R$, and hence $(A_1, \ldots, A_{k-1}, C, A_{k+1}, \ldots A_n)$ is weakly (ultra-weakly) comm-safe. □

3.2. Pairwise and Incremental Analysis. The global property of weak comm-safety can be derived from pairwise compatibility analysis. In contrary, to verify ultra-weak comm-safety one needs to apply an incremental approach: Knowing comm-safety for $n-1$ transition systems, it suffices to check compatibility between their composition and an nth transition system to derive comm-safety for the complete product. In the following we make these claims precise.

Theorem 4.37 (Pairwise weak) *Let $A_1, \ldots, A_n$ be pairwise composable PIOs. If $A_i \leftrightarrow_w A_j$ for all $i, j \in \{1, \ldots, n\}$ with $i \neq j$, then $(A_1, \ldots, A_n)$ is weakly comm-safe.*

PROOF. By induction on the number of composed transition systems.

Base case ($n = 2$). Let (A_1, A_2) be pairwise composable PIOs and assume $A_1 \leftrightarrow_w A_2$, then, by the conditions for weak output compatibility, for all $(s_{A_1}, s_{A_2}) \in \mathscr{R}(A_1 \otimes A_2)$,

$$\forall l \in O_{A_1} \cap I_{A_2} \,.\, \forall s'_{A_1} \in S_{A_1} \,.\, (s_{A_1}, l, s'_{A_1}) \in \Delta_{A_1} \implies (\exists s'_{A_2} \in S_{A_2} \,.\, (s_{A_2}, l, s'_{A_2}) \in \Pi^{\mathcal{T}(A_2)}_{A_2}) \tag{1}$$

$$\forall l \in O_{A_2} \cap I_{A_1} \,.\, \forall s'_{A_2} \in S_{A_2} \,.\, (s_{A_2}, l, s'_{A_2}) \in \Delta_{A_2} \implies (\exists s'_{A_1} \in S_{A_1} \,.\, (s_{A_1}, l, s'_{A_1}) \in \Pi^{\mathcal{T}(A_1)}_{A_1}) \tag{2}$$

By (1) it follows that (s_{A_1}, s_{A_2}) is weakly comm-safe for A_1 and by (2) the same holds for A_2, thus (A_1, A_2) is weakly comm-safe.

Induction step ($n{+}1$). Let $(A_1, \ldots, A_{n+1})$ be pairwise composable and weakly output compatible PIOs. Let $A_R = (A_1 \otimes \cdots \otimes A_n)$. By the assumption of pairwise compatibility, $A_{n+1} \leftrightarrow_w A_i$ for all $1 \leq i \leq n$, thus it follows by Lem. 4.38 that $A_{n+1} \leftrightarrow_w A_R$. Thus, by induction assumption, (A_R, A_{n+1}) is weakly comm-safe and therefore $(A_1, \ldots, A_{n+1})$ is weakly comm-safe. □

Lemma 4.38 *Let (A_1, A_2, A_3) be pairwise composable PIOs. If $A_1 \leftrightarrow_w A_2$, $A_1 \leftrightarrow_w A_3$ and $A_2 \leftrightarrow_w A_3$, then $A_1 \leftrightarrow_w (A_2 \otimes A_3)$ and $(A_1 \otimes A_3) \leftrightarrow_w A_2$.*

PROOF. First we show $A_1 \leftrightarrow_w (A_2 \otimes A_3)$. By $A_1 \leftrightarrow_w A_2$, the conditions (1) and (2) above (Proof (Prop. 4.37), p. 65) hold. Let $(s_{A_1}, s_{A_2}, s_{A_3}) \in \mathscr{R}(A_1 \otimes A_2 \otimes A_3)$. Obviously we have $(s_{A_1}, s_{A_2}) \in \mathscr{R}(A_1 \otimes A_2)$ and $(s_{A_2}, s_{A_3}) \in \mathscr{R}(A_2 \otimes A_3)$. By (1) there exists $\bar{s}_{A_2}, \bar{s}'_{A_2} \in S_{A_2}$ such that $(s_{A_2}, \bar{s}_{A_2}) \in \Pi^{\mathcal{T}(A_2)}_{A_2}$ and $(\bar{s}_{A_2}, l, \bar{s}'_{A_2}) \in \Pi_{A_2}$. Since $\mathcal{T}(A_2)$ is not shared with any composable PIO, Lem. 4.3 is applicable and it follows that $(\bar{s}_{A_2}, s_{A_3}) \in \Pi^{\mathcal{T}(A_2)}_{A_2 \otimes A_3}$ and thus condition (A-Out) of weak output compatibility holds for A_1 and $A_2 \otimes A_3$. Condition (B-Out) holds analogously using (2) and hence $A_1 \leftrightarrow_w (A_2 \otimes A_3)$.

Since we could have chosen A_2 instead of A_1 and $A_1 \otimes A_3$ instead of $A_2 \otimes A_3$ we have also $(A_1 \otimes A_3) \leftrightarrow_w A_2$. □

Example 4.39 (Pairwise ultra-weak) *Proposition 4.37 does not hold for the case of ultra-weak output compatibility. Let A_1, A_2 and A_3 be pairwise composable PIOs as given in Fig. 4.12. Let $\mathcal{S}(A_1, A_2) = \{x\}$, $\mathcal{S}(A_2, A_3) = \{y\}$ and $\mathcal{S}(A_1, A_3) = \{z\}$.*

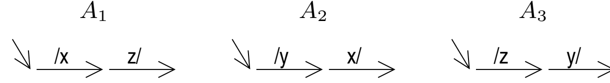


FIGURE 4.12. Pairwise ultra-weak compatibility does not imply comm-safety

Then $A_1 \leftrightarrow_u A_2$, $A_2 \leftrightarrow_u A_3$ and $A_1 \leftrightarrow_u A_3$ but (A_1, A_2, A_3) is not ultra-weakly comm-safe, due to state $(s_{0,A_1}, s_{0,A_2}, s_{0,A_3})$ which is globally reachable but invalidates the requirement of ultra-weak comm-safety (Def. 4.34) for A_k with any $k \in \{1, 2, 3\}$.

As the example suggests we need a kind of global analysis to obtain ultra-weak communication safety. This kind of analysis is also valid for weak compatibility.

Proposition 4.40 (Increment. ultra-weak) *Let $(A_1, \ldots, A_n)$ be pairwise composable PIOs. If $(A_1, \ldots A_{n-1})$ is ultra-weakly comm-safe and $(\otimes_{i=1}^{n-1} A_i) \leftrightarrow_u A_n$ then $(A_1, \ldots, A_n)$ is ultra-weakly comm-safe.*

PROOF. By induction on the number n of PIOs. The case $n = 0$ is trivial. Let $n \in \mathbb{N}$. By induction hypothesis it holds that $(A_1, \ldots A_n)$ is ultra-weakly comm-safe. We need to show that $(\otimes_{i=1}^{n} A_i) \leftrightarrow_u A_{n+1}$ implies ultra-weak comm-safety of $(A_1, \ldots, A_{n+1})$. Let $A_R = (\otimes_{i=1}^{n} A_i)$. By the compatibility condition (A-Out) for $A_R \leftrightarrow_u A_{n+1}$ we obtain that A_{n+1} is ultra-weakly comm-safe in $(A_1, \ldots, A_{n+1})$. With condition (B-Out) we obtain ultra-weak comm-safety of A_R in $(A_1, \ldots, A_{n+1})$. □

Note that the reverse direction considered in Lem. 4.36 does not claim communication-safety of $(A_1, \ldots A_{n-1})$. Compatibility within a composition of transition systems may depend on "unused inputs". For example an output which is preceeded by an input that is shared but not used by the communication partner is not reachable in the product of these transition systems. Therefore, even though these transition systems are incompatible their composition results in a transition system that is comm-safe. Unused input transitions motivated the "optimistic" approach to compatibility studied by de Alfaro and Henzinger [**dAH01a**]. The idea is to allow for incompatibilities as long as these are reachable only via unused inputs. Their approach, however requires a special composition operator.

4. Greybox Refinement and General Properties

Within this section we study a notion of refinement, greybox refinement, that allows to take synchronisation transitions within compositions of transition systems into account. The difference with regard to blackbox refinement is similar to the difference between

(A-IOT) $\forall l \in I_A \cup O_A \cup T_A \,.\, \forall s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Pi_A \implies$
$(\exists s'_C \in S_C \,.\, (s_C, l, s'_C) \in \Pi^\tau_C \wedge (s'_A, s'_C) \in R)$

(A-Tau) $\forall s'_A \in S_A \,.\, (s_A, \tau, s'_A) \in \Pi_A \implies$
$(\exists s'_C \in S_C \,.\, (s_C, s'_C) \in \Pi^\tau_C \wedge (s'_A, s'_C) \in R)$

(C-IOT) $\forall l \in I_C \cup O_C \cup T_C \,.\, \forall s'_C \in S_C \,.\, (s_C, l, s'_C) \in \Delta_C \implies$
$(\exists s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta^\tau_A \wedge (s'_A, s'_C) \in R)$

(C-Tau) $\forall s'_C \in S_C \,.\, (s_C, \tau, s'_C) \in \Delta_C \implies$
$(\exists s'_A \in S_A \,.\, (s_A, s'_A) \in \Delta^\tau_A \wedge (s'_A, s'_C) \in R)$

FIGURE 4.13. Conditions for greybox refinement between A and C

greybox and blackbox equivalence considered in the context of I/O-transition systems used for the formalisation of component behaviours in Chap. 2. The definition treats internal transitions like input and output, but is still relaxed with regard to τ-transitions.

Definition 4.41 (Greybox refinement) *Let A and C be PIOs. C is a* greybox refinement *of A, written $C \sqsubseteq^{\mathrm{gb}} A$, if $L_A = L_C$ and there exists $R \subseteq S_A \times S_C$ such that $(s_{0,A}, s_{0,C}) \in R$ and for all $(s_A, s_C) \in R$ the conditions in Tab. 4.13 hold.*

Greybox refinement strengthens the conditions for internal transitions to be simulated analogously to I/O-transitions. Thus, a given greybox refinement relation can be used to witness blackbox refinement. Furthermore, greybox and blackbox refinement coincide for systems without internal transitions.

Lemma 4.42 *Let A and C be PIOs with $L_A = L_C$. If $C \sqsubseteq^{\mathrm{gb}} A$ then $C\xi \sqsubseteq^{\mathrm{gb}} A\xi$.* □

Proposition 4.43 *Let A and C be PIOs with $L_A = L_C$. Then the following holds.*
(1) If $C \sqsubseteq^{\mathrm{gb}} A$ then $C \sqsubseteq^{\mathrm{bb}} A$.
(2) If $C \sqsubseteq^{\mathrm{bb}} A$ and $T_A = T_C = \emptyset$ then $C \sqsubseteq^{\mathrm{gb}} A$.

PROOF. (1) By Lem. 4.42 it holds that $C\xi \sqsubseteq^{\mathrm{gb}} A\xi$. Consider the conditions of $\sqsubseteq^{\mathrm{gb}}$. If we hide internal transitions in A and C, then we may remove T_A and add T_C in condition (A-IOT) such that condition (A-I/O) of blackbox refinement is obtained. The remaining conditions are treated analogously and hence, for A and C with hidden internal transitions, the conditions (A-IOT), (A-Tau), (C-IOT) and (C-Tau) are equivalent to the respective conditions (A-I/O), (A-Int), (C-I/O) and (C-Int) of blackbox refinement. (2) For PIOs without internal transitions, the conditions for greybox and blackbox refinement coincide analogously to (1). For instance, adding T_A and removing T_C from condition (A-I/O) of blackbox refinement yields (A-IOT); and similar for the remaining clauses. □

Note that the assumption on empty sets of internal transitions is necessary. Consider for example the case of $T_A \neq \emptyset$. Then C is allowed by clause (A-Int) of blackbox refinement to skip internal transitions labelled with $l \in T_A$. Such an internal transition, however, would be required to be preserved along clause (A-IOT) of greybox refinement. If, on the other hand, $T_C \neq \emptyset$ then it would be possible for the concrete system C to add an internal transition not present in A along clause (C-Int) of blackbox refinement. But this is not compatible with requirement (C-IOT) of greybox refinement.

Proposition 4.44 (Preorder) *Let A, C and D be PIOs. Then $A \sqsubseteq^{\mathrm{gb}} A$ and if $D \sqsubseteq^{\mathrm{gb}} C$ and $C \sqsubseteq^{\mathrm{gb}} A$ then $D \sqsubseteq^{\mathrm{gb}} A$.*

PROOF. Analogously to blackbox refinement (cf. Prop. 4.8). □

Theorem 4.45 (Precongruence) *Let A, B and C be PIOs such that A, B and C, B are composable. If $C \sqsubseteq^{\mathrm{gb}} A$, then $C \otimes B \sqsubseteq^{\mathrm{gb}} A \otimes B$.*

PROOF. Let $A \otimes B = ((I_{AB}, O_{AB}, T_{AB}), S_{AB}, s_{0,AB}, \Delta_{AB}, \Pi_{AB})$ and $C \otimes B = ((I_{CB}, O_{CB}, T_{CB}), S_{CB}, s_{0,CB}, \Delta_{CB}, \Pi_{CB})$. Let R_{AC} be a witness for $C \sqsubseteq^{\mathrm{gb}} A$ with $(s_{0,A}, s_{0,C}) \in R_{AC}$ and let

$$R = \{((s_A, s_B), (s_C, s_B)) \mid (s_A, s_C) \in R_{AC} \wedge (s_A, s_B) \in \mathscr{R}(A \otimes B)\},$$

then $((s_{0,A}, s_{0,B}), (s_{0,C}, s_{0,B})) \in R$. Let $((s_A, s_B), (s_C, s_B)) \in R$. We check the conditions of greybox refinement for R.

(Case A-IOT) Let $l \in I_{AB} \cup O_{AB} \cup T_{AB}$ and $((s_A, s_B), l, (s'_A, s'_B)) \in \Pi_{AB}$.
If $l \in I_A \cup O_A \cup T_A$, then there exists $(s_A, l, s'_A) \in \Pi_A$ and $s_B = s'_B$. By clause (A-IOT) for R_{AC} there exists $s'_C \in S_C$ such that $(s_C, l, s'_C) \in \Pi^\tau_C$ and $(s'_A, s'_C) \in R_{AC}$. Since $l \notin \mathcal{S}(C, B)$ (by $l \notin \mathcal{S}(A, B)$, $L_A = L_C$) we have by Lem. 4.3 (2), that $((s_C, s_B), l, (s'_C, s_B)) \in \Pi^\tau_{CB}$. Since $(s'_A, s_B) \in \mathscr{R}(A \otimes B)$ it follows that $((s'_A, s_B), (s'_C, s_B)) \in R$.
If $l \in I_B \cup O_B \cup T_B$, then there exists $(s_B, l, s'_B) \in \Pi_B$ and $s_A = s'_A$. Hence $((s_C, s_B), l, (s_C, s'_B)) \in \Pi_{CB}$ and since $(s_A, s'_B) \in \mathscr{R}(A \otimes B)$, it follows that $((s_A, s'_B), (s_C, s'_B)) \in R$.
If $l \in \mathcal{S}(A, B)$, then there exists $(s_A, l, s'_A) \in \Pi_A$ and $(s_B, l, s'_B) \in \Pi_B$, and either $l \in I_A \cap O_B$ or $l \in O_A \cap I_B$. In both cases, by (A-IOT) for R_{AC}, there exists $s'_C \in S_C$ such that $(s_C, l, s'_C) \in \Pi^\tau_C$ and $(s'_A, s'_C) \in R_{AC}$. By Lem. 4.3 (3) it follows that $((s_C, s_B), l, (s'_C, s'_B)) \in \Pi^\tau_{CB}$ and since $(s'_A, s'_B) \in \mathscr{R}(A \otimes B)$ we have $((s'_A, s'_B), (s'_C, s'_B)) \in R$.

(Case A-Tau) Let $((s_A, s_B), \tau, (s'_A, s'_B)) \in \Pi_{AB}$. Then either $(s_A, \tau, s'_A) \in \Pi_A$ and $s_B = s'_B$ or $(s_B, \tau, s'_B) \in \Pi_B$ and $s_A = s'_A$. In the former case, by (A-Tau) for R_{AC}, there exists $s'_C \in S_C$ such that $(s_C, s'_C) \in \Pi^\tau_C$ and $(s'_A, s'_C) \in R_{AC}$. Then, by Lem. 4.3 (1), $((s_C, s_B), (s'_C, s_B)) \in \Pi^\tau_{CB}$. Since $(s'_A, s_B) \in \mathscr{R}(A \otimes B)$ it follows that $((s'_A, s_B), (s'_C, s_B)) \in R$. For the latter case we have $((s_C, s_B), \tau, (s_C, s'_B)) \in \Pi_{CB}$ and since $(s_A, s'_B) \in \mathscr{R}(A \otimes B)$ it follows that $((s_A, s'_B), (s_C, s'_B)) \in R$.

(Case C-IOT) Let $l \in I_{CB} \cup O_{CB} \cup T_{CB}$ and $((s_C, s_B), l, (s'_C, s'_B)) \in \Pi_{CB}$.
If $l \in I_C \cup O_C \cup T_C$, then there exists $(s_C, l, s'_C) \in \Delta_C$ and $s_B = s'_B$. By clause (C-IOT) for R_{AC} there exists $s'_A \in S_A$ such that $(s_A, l, s'_A) \in \Delta^\tau_A$ and $(s'_A, s'_C) \in R_{AC}$. By Lem. 4.3 (2) it follows that $((s_A, s_B), l, (s'_A, s_B)) \in \Delta^\tau_{AB}$. Since $(s'_A, s_B) \in \mathscr{R}(A \otimes B)$ we also have $((s'_A, s_B), (s'_C, s_B)) \in R$.
If $l \in I_B \cup O_B \cup T_B$, then there exists $(s_B, l, s'_B) \in \Delta_B$ and $s_C = s'_C$. Hence $((s_A, s_B), l, (s_A, s'_B)) \in \Delta_{AB}$ and since $(s_A, s'_B) \in \mathscr{R}(A \otimes B)$ and $(s_A, s_C) \in R_{AC}$ it follows that $((s_A, s'_B), (s_C, s'_B)) \in R$.
If $l \in \mathcal{S}(C, B)$, then there exists $(s_C, l, s'_C) \in \Delta_C$ and $(s_B, l, s'_B) \in \Delta_B$, and either $l \in I_C \cap O_B$ or $l \in O_C \cap I_B$. In both cases, by (C-IOT) for R_{AC}, there exists $s'_A \in S_A$ such that $(s_A, l, s'_A) \in \Delta^\tau_A$ and $(s'_A, s'_C) \in R_{AC}$. By Lem. 4.3 (3) it follows that $((s_A, s_B), (s'_A, s'_B)) \in \Delta^\tau_{AB}$, hence $(s'_A, s'_B) \in \mathscr{R}(A \otimes B)$ and $((s'_A, s'_B), (s'_C, s'_B)) \in R$.

(Case C-Tau) Let $((s_C, s_B), \tau, (s'_C, s'_B)) \in \Delta_{CB}$. Then either $(s_C, \tau, s'_C) \in \Delta_C$ and $s_B = s'_B$, or $(s_B, \tau, s'_B) \in \Delta_B$ and $s_C = s'_C$. In the former case, by (C-Tau) for R_{AC} there exists $s'_A \in S_A$ such that $(s_A, s'_A) \in \Delta^\tau_A$ and $(s'_A, s'_C) \in R_{AC}$. Then by Lem. 4.3 (1), $((s_A, s_B), (s'_A, s_B)) \in \Delta^\tau_{AB}$. Since $(s'_A, s_B) \in \mathscr{R}(A \otimes B)$ it follows that $((s'_A, s_B), (s'_C, s_B)) \in R$. For the latter case we have $((s_A, s_B), \tau, (s_A, s'_B)) \in \Delta_{AB}$ and since $(s_A, s'_B) \in \mathscr{R}(A \otimes B)$ it follows that $((s_A, s'_B), (s_C, s'_B)) \in R$. ☐

Corollary 4.46 (Compositionality) *Let A, B and C, D be PIOs such that A, B and C, D are composable. If $C \sqsubseteq^{\mathrm{gb}} A$ and $D \sqsubseteq^{\mathrm{gb}} B$, then $C \otimes D \sqsubseteq^{\mathrm{gb}} A \otimes B$.*

PROOF. The proof is analogous to the case of blackbox refinement (cf. Prop. 4.12) using Prop. 4.44 and Thm. 4.45 for greybox refinement. ☐

4.1. Blackbox vs Greybox Refinement. Blackbox refinement between PIOs without internal transitions can be used to derive greybox refinement.

Theorem 4.47 (Blackbox vs Greybox) *Let A, B and C be PIOs such that A, B are composable with $T_A = T_B = T_C = \emptyset$. If $C \sqsubseteq^{\mathrm{bb}} A$ then $C \otimes B \sqsubseteq^{\mathrm{gb}} A \otimes B$.*

PROOF. The claim follows by Prop. 4.43 (2) and Thm. 4.45. □

Using Cor. 4.46 we obtain from corresponding assumptions that $C \otimes D \sqsubseteq^{\mathrm{gb}} A \otimes B$. Therefore, in order to establish greybox refinement between compositions of PIOs, one either checks greybox refinement for each of the PIOs of the composition or, in case there are local internal transitions that hinder this approach, one first hides all local steps and then checks either blackbox or greybox refinement.

For asynchronous communication we consider only the case of completely buffered PIOs. The general case holds analogously.

Theorem 4.48 (Blackbox vs Greybox (asynchronous)) *Let A, B and C be PIOs such that A, B and C, B are composable. If $C \sqsubseteq^{\mathrm{bb}} A$ then $\Omega(C)\xi \otimes \Omega(B)\xi \sqsubseteq^{\mathrm{gb}} \Omega(A)\xi \otimes \Omega(B)\xi$.*

PROOF. By Thm. 4.25 we have $\Omega(C) \sqsubseteq^{\mathrm{bb}} \Omega(A)$ and by Lem. 4.5, $\Omega(C)\xi \sqsubseteq^{\mathrm{bb}} \Omega(A)\xi$. Now the claim follows by Prop. 4.47. □

It is an open question if Thm. 4.48 holds for buffered PIOs where only the local actions of the underlying control automata A, B and C are hidden.

4.2. Weak Simulation and Temporal Logics. The conditions (C-IOT) and (C-Tau) of greybox refinement correspond to the requirements of weak simulation between general labelled transition systems. By this means it is immediate that properties expressed with respect to the transition relation Δ_A of an abstract PIO A which are known to be preserved by weak simulation are also preserved by greybox refinement, and thus hold for a greybox refinement C of A. In the following we will define a notion of weak simulation for PIOs that takes only the relation Δ into account and investigate preservation results based on finite and infinite weak traces. A specialised logic that takes both relations into account is defined in [**HJS01**], though only for strong modal logic.

Definition 4.49 (Weak simulation) *Let A and C be PIOs. A* weakly simulates *C, written $C \preccurlyeq_\Delta A$, if $L_A = L_C$ and there exists $R \subseteq S_A \times S_C$ such that $(s_{0,A}, s_{0,C}) \in R$ and for all $(s_A, s_C) \in R$ the following conditions hold:*

(C-IOT) $\forall l \in I_C \cup O_C \cup T_C \,.\, \forall s'_C \in S_C \,.\, (s_C, l, s'_C) \in \Delta_C \implies$
$(\exists s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta^\tau_A \wedge (s'_A, s'_C) \in R)$

(C-Tau) $\forall s'_C \in S_C \,.\, (s_C, \tau, s'_C) \in \Delta_C \implies$
$(\exists s'_A \in S_A \,.\, (s_A, s'_A) \in \Delta^\tau_A \wedge (s'_A, s'_C) \in R)$

Analogously, we define a weak simulation from C to A, denoted by $A \preccurlyeq_\Pi C$ using the set of persistent transitions Π. In the following we focus on the direction from A to C. Weak simulation from C to A is used in the discussion of liveness properties below.

Corollary 4.50 (Weak simulation) *Let A and C be PIOs. If $C \sqsubseteq^{\mathrm{gb}} A$ then $C \preccurlyeq_\Delta A$.*

With Cor. 4.50 a number of known results for weak simulation carry over to greybox refinement. In the case of τ-free transition systems one could even transfer results known for strong simulation relations, for instance preservation of the universal and the existential fragments of CTL* [**BKL08**, Thm. 7.76, Thm. 7.79]. In the following, however, we stick to transition systems with τ-transitions.

First, we consider trace inclusion. Let A be a PIO and let $s \in S_A$. The state s is *terminal* if it has no outgoing transitions, i.e. for all $l \in \mathcal{L}(A)$ and for all $s' \in S_A$ it holds that $(s, l, s') \notin \Delta_A$. A *finite* Δ*-run* of A is a finite sequence $s_{0,A} l_1 s_1 \dots s_{n-1} l_n s_n$ with $s_i \in S_A$, $l_{i+1} \in \bigcup L_A$ and $(s_i, l_{i+1}, s_{i+1}) \in \Delta_A^\tau$ for all $0 \leq i < n$. An *infinite* Δ*-run* of A is an infinite sequence $s_{0,A} l_1 s_1 l_2 \dots$ with $s_i \in S_A$, $l_{i+1} \in \bigcup L_A$ and $(s_i, l_{i+1}, s_{i+1}) \in \Delta_A^\tau$ for all $i \geq 0$. A *maximal* Δ*-run* of A is either infinite, or finite with s_n being a terminal state.

A *finite weak* Δ*-trace* of A is a finite sequence $l_1 \dots l_n$ with $l_i \in \bigcup L_A$ such that there is a finite Δ-run $s_0 l_1 s_1 \dots s_{n-1} l_n s_n$ of A. With $\mathscr{T}_\Delta^*(A)$ we denote the set of finite weak Δ-traces of A. An *infinite weak* Δ*-trace* of A is an infinite sequence $l_1 l_2 \dots$ with $l_i \in \bigcup L_A$ such that there is a infinite Δ-run $s_0 l_1 s_1 l_2 \dots$ of A. With $\mathscr{T}_\Delta(A)$ we denote the set of all infinite and finite weak Δ-traces such that there is a maximal Δ-run of A.

Proposition 4.51 (Greybox trace inclusion) *Let A and C be PIOs. Let C be without terminal states. If $C \sqsubseteq^{\mathrm{gb}} A$ then $\mathscr{T}_\Delta(C) \subseteq \mathscr{T}_\Delta(A)$.*

PROOF. Since C is without terminal states it suffices to consider infinite runs. Let $\varrho = s_{0,C} l_1 s_{1,C} l_2 \dots$ be an infinite Δ-run of C. By Cor. 4.50 there exists a weak simulation relation $R \subseteq S_A \times S_C$ such that $(s_{i,A}, s_{i,C}) \in R$ for all $i \geq 0$. The relation can be used to construct inductively an infinite Δ-run $s_{0,A} l_1 s_{1,A} l_2 \dots$ of A. Since the Δ-runs yield the same infinite weak Δ-trace, we have $\mathscr{T}_\Delta(C) \subseteq \mathscr{T}_\Delta(A)$. □

The assumption on terminal states is necessary. The example in Fig. 4.14 satisfies $C \sqsubseteq^{\mathrm{gb}} A$ as witnessed by the relation $R = \{(a_1, c_1), (a_2, c_2), (a_3, c_3), (a_4, c_2)\}$. But $\mathscr{T}_\Delta(C) \not\subseteq \mathscr{T}_\Delta(A)$ since $\mathsf{m}_1 \in \mathscr{T}_\Delta(C)$ and $\mathsf{m}_1 \notin \mathscr{T}_\Delta(A)$.

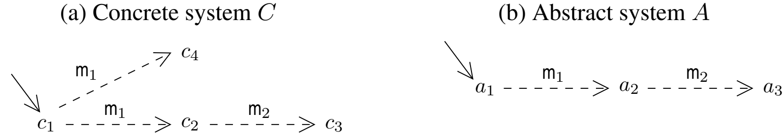


FIGURE 4.14. Terminal states invalidate greybox trace inclusion

Greybox trace inclusion also holds for finite weak Δ-traces.[2] We use Prop. 4.51 for the proof of the following lemma and hence assume C to be without terminal states. A direct proof would not need this assumption. Let $\varrho \in \mathscr{T}_\Delta(A)$ be a weak infinite Δ-trace of A. The set of *finite prefixes* $\mathit{pref}(\varrho)$ is given by $\mathit{pref}(\varrho) = \{\bar{\varrho} \in (\bigcup L_A)^* \mid \bar{\varrho} \text{ is a finite prefix of } \varrho\}$. For example, if $\varrho = l_1 l_2 \dots$ then $\mathit{pref}(\varrho) = \{\epsilon, l_0, l_0 l_1, \dots\}$.

Lemma 4.52 *Let A and C be PIOs. Let C be without terminal states. If $C \sqsubseteq^{\mathrm{gb}} A$ then $\mathscr{T}_\Delta^*(C) \subseteq \mathscr{T}_\Delta^*(A)$.*

PROOF. By Prop. 4.51 we have $\mathscr{T}_\Delta(C) \subseteq \mathscr{T}_\Delta(A)$. The set of finite prefixes for $\mathscr{T}_\Delta(A)$ and $\mathscr{T}_\Delta(C)$ coincide with the set of finite weak Δ-traces of A and C respectively. Since pref is monotone we have $\mathit{pref}(\mathscr{T}_\Delta(C)) \subseteq \mathit{pref}(\mathscr{T}_\Delta(A))$, where $\mathit{pref}(\mathscr{T}_\Delta(C)) = \bigcup_{\varrho \in \mathscr{T}_\Delta(C)} \mathit{pref}(\varrho)$ and analogous for A. Therefore $\mathscr{T}_\Delta^*(C) \subseteq \mathscr{T}_\Delta^*(A)$. □

The infinite part of greybox trace inclusion was used in [**HJK09**] to prove the preservation of buffered compatibility, a notion of compatibility similar to our current definition of asynchronous compatibility. In the following we investigate the preservation of more general properties.

Safety and liveness properties. Safety properties specify that, within a transition system, a certain action may not happen or a certain state may not be reachable. In the following we

[2] cf. [**BKL08**, Thm. 3.30] for a corresponding claim w.r.t. strong traces.

consider the formalisation of safety properties as given by Tracta in the context of labelled transition systems (LTS) [**Gia99**]. Tracta focuses on actions and uses finite deterministic τ-free labelled transition systems for the specification of safety properties.[3] The finite traces of these *property automata* formally represent a safety property P using a subset of the observable alphabet of the LTS to which P relates. Satisfaction is defined using finite trace inclusion between the property P and the LTS restricted to the alphabet of P. Tracta uses traces with *observable* actions [**Gia99**, p. 14] corresponding to our notion of weak finite traces (cf. $\mathscr{T}^*$ for IOTSs in Sect. 2.1 in Chap. 3). Based on this observation we transfer the Tracta approach to our setting as follows.

Denote the weak finite traces of an LTS by $\mathscr{T}^*$. Let A be a PIO. A *safety property* $P_{\rm sf}$ for A is specified by a τ-free deterministic LTS $P_{\rm sf} = (\bigcup L_A, S, s_0, \Delta)$. *$A$ satisfies* $P_{\rm sf}$, written $A \models P_{\rm sf}$ if and only if $\mathscr{T}^*_\Delta(A/H) \subseteq \mathscr{T}^*(P_{\rm sf})$, where $H = \bigcup L_A \setminus \bigcup L_{P_{\rm sf}}$.

Next we show that greybox refinement preserves safety properties. For the proof we rely on Lem. 4.52 which requires the concrete system to be without terminal states. Using Cor. 4.50 and a lemma for PIOs analogous to Lem. 3.6 on finite trace equivalence between IOTSs one could drop this assumption for the following theorem.

Theorem 4.53 *Let A and C be PIOs. Let C be without terminal states. If $C \sqsubseteq^{\rm gb} A$ then it holds that $A \models P_{\rm sf}$ implies $C \models P_{\rm sf}$.*

PROOF. By $A \models P_{\rm sf}$ we have $\mathscr{T}^*_\Delta(A/H) \subseteq \mathscr{T}^*(P_{\rm sf})$ with $H = \bigcup L_A \setminus \bigcup L_{P_{\rm sf}}$. Since $H \subseteq L_A$ and $L_A = L_C$ by $C \sqsubseteq^{\rm gb} A$, it follows from the definitions that $C/H \sqsubseteq^{\rm gb} A/H$. Moreover, hiding does not affect the absence of terminal states, thus Lem. 4.52 is applicable and it follows that $\mathscr{T}^*_\Delta(C/H) \subseteq \mathscr{T}^*_\Delta(A/H)$ and hence $C \models P_{\rm sf}$. $\square$

There is an interesting connection between the satisfaction of safety properties and neutral behaviours as discussed in Chap. 3. Since property automata are τ-free we may apply Prop. 3.8 in order to check if a certain safety property is satisfied. Neutrality was considered in the context of component behaviours and IOTSs. A corresponding notion for PIOs could be defined with regard to the transition relation Δ. Let A and B be composable PIOs. *B is Δ-neutral for A* if $A\theta_{\mathcal{S}(A,B)} \approx^{\rm gb}_\Delta A \otimes B$ (cf. p. 36), where $\approx^{\rm gb}_\Delta$ denotes greybox equivalence with respect to the transition relations Δ on both sides. With an analogy to Prop. 3.8 we obtain for composable PIOs A and B such that $\bigcup L_B \subseteq \bigcup L_A$ and $H = \bigcup L_A \setminus \bigcup L_B$, B is Δ-neutral for A iff $\mathscr{T}^*_\Delta(A/H) \subseteq \mathscr{T}^*(B)$. Therefore, if $P_{\rm sf}$ is a safety property for A we have

$$A \models P_{\rm sf} \text{ if and only if } P_{\rm sf} \text{ is } \Delta\text{-neutral for } A.$$

Liveness properties assert that "eventually something good happens". In contrast to safety properties along finite traces, eventuality relates to the infinite traces of a system and requires, in the context of transition systems, an action to happen infinitely often or a state to be visited repeatedly, for instance. Using a weak simulation from C to A, the results above hold analogously for traces in Π. Therefore, we could use $A \preccurlyeq_\Pi C$ to claim the preservation of liveness properties similar to Thm. 4.53. However, as weak simulation is known to be ignorant with regard to divergence (infinite τ-paths) it is immediate that greybox refinement does not preserve liveness properties in general. Consider for instance the two examples in Fig. 4.15, taken from [**Gia99**, p. 101]. A liveness property ϕ could require that it is always the case that if m happens then eventually n happens, using LTL the requirement reads as $\Box(\mathsf{m} \implies \Diamond\mathsf{n})$. Both abstract PIOs, A_1 and A_2, satisfy ϕ, but neither does C_1 nor C_2, which is due to the diverging τ-paths.

One way to cope with τ loops is to extend greybox refinement with an additional requirement for divergence, making the refinement relation divergence-sensitive. By this means liveness properties such as ϕ could be preserved. We believe that such an extension

[3] Determinism is not a restriction. A non-deterministic LTSs can be transformed to an equivalent deterministic LTS; cf. [**Gia99**, Sect. 5.1.3]

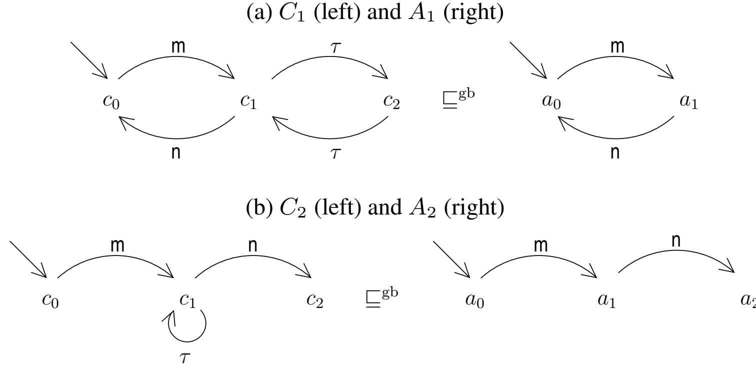


FIGURE 4.15. Greybox refinement ignores divergence

would allow for an even more general statement, as divergence-sensitive weak equivalence is known to preserve $\text{CTL}^*_{\setminus\bigcirc}$ formulas [**BKL08**, Lem. 7.129], that is, CTL^* formulas without the next operator. Since CTL^* comprises both, CTL and LTL, such an extension would be quite useful. We briefly address this issue in the next section.

Stutter trace inclusion. Greybox trace inclusion in the context of PIOs seems to be an analogy to stutter trace inclusion in the context of Kripke structures as considered by [**BKL08**, p. 533].[4] In this case, Cor. 7.93 of [**BKL08**] would carry over which states the preservation of $\text{LTL}_{\setminus\bigcirc}$ formulas. Stutter steps are used to abstract from internal steps. Stutter trace inclusion then requires for all traces ϱ_C of a concrete system C the existence of a trace ϱ_A in the abstract system A such that ϱ_C and ϱ_A are stutter equivalent, i.e. equivalent modulo stutter steps. However, a closer look reveals an important difference in the underlying set of traces. [**BKL08**, Def. 7.89] considers traces as given by the executions (runs) of the underlying transition system. Our formalisation, in contrast, uses *weak* traces without internal labels, that is, traces that rely on steps $(s_i, l_{i+1}, s_{i+1}) \in \Delta^\tau$ with $l_i \in \bigcup L$. A definition that includes internals would require $l_i \in \bigcup L \cup \{\tau\}$.

More precisely, an *infinite strong* Δ*-run* of a PIO A is an infinite sequence $s_{0,A} l_1 s_1 l_2 \ldots$ with $s_i \in S_A$, $l_{i+1} \in \bigcup L_A \cup \{\tau\}$ and if $l_{i+1} \neq \tau$ then $(s_i, l_{i+1}, s_{i+1}) \in \Delta^\tau_A$, and if $l_{i+1} = \tau$ then $(s_i, s_{i+1}) \in \Delta^\tau_A$, for $i \geq 0$. An *infinite strong* Δ*-trace* of A is an infinite sequence $l_1 l_2 \ldots$ with $l_i \in \bigcup L_A \cup \{\tau\}$ such that there is a strong Δ-run $s_0 l_1 s_1 l_2 \ldots$ of A. We denote the set of infinite strong Δ-traces of A by $\mathscr{T}^s_\Delta(A)$. Note that the definition is still relaxed with respect to the number of τ transitions.

As mentioned above, greybox refinement is too weak with regard to divergent τ paths to ensure trace inclusion for strong Δ-traces. Consider for example the transition systems in Fig. 4.16. It holds that $C \sqsubseteq^{\text{gb}} A$ but $\mathscr{T}^s_\Delta(C) \not\subseteq \mathscr{T}^s_\Delta(A)$ since $\tau^\omega = \tau\tau\ldots \in \mathscr{T}^s_\Delta(C)$ while $\tau^\omega \notin \mathscr{T}^s_\Delta(A)$.

FIGURE 4.16. Greybox refinement and strong Δ-traces

[4] In other words, greybox trace inclusion uses the action (event) labels of a transition system, hence is action-based, while stutter trace inclusion uses state propositions, hence is state-based.

A solution along the lines of divergence-sensitive equivalence relations as defined in [**BKL08**, Def. 7.106] requires an extension of our refinement relations that takes divergence of the concrete system into account. A state $s \in S_A$ of a PIO A *diverges* if there exists an infinite execution fragment $sl_1s_1l_2\ldots \in \mathit{frag}(A)$ such that $l_i = \tau$ for all $i \geq 1$. Let A and C be PIOs. A refinement relation $R \subseteq S_A \times S_C$ is *divergence-sensitive* if for all $(s_A, s_C) \in R$ it holds that, if s_C diverges then s_A diverges. A definition of *divergence-sensitive greybox refinement* requires the existence of a divergence-sensitive relation $R \subseteq S_A \times S_C$ such that $(s_{0,A}, s_{0,C}) \in R$ and for all $(s_A, s_C) \in R$ the conditions in Tab. 4.13 hold.

Since divergence-sensitive weak equivalence is known to preserve $\mathrm{CTL}^*_{\setminus\bigcirc}$ formulas we believe that it is possible to show an analogous result for divergence-sensitive greybox refinement. Such a result requires a translation between PIOs and Kripke structures which could be based on work of de Nicola and Vaandrager who define a mapping between labelled transition systems and Kripke structures in [**NV90**]. However, still this relates to the slightly restricted setting of this section which considered only the Δ part of PIOs and therefore it could be more effective to approach directly an extension of the strong modal logic developed by [**HJS01**] to a weak modal logic suitable for the specification of properties with regard to both transition relations of a PIO.

5. Discussion and Related Work

The development of a refinement theory initially started with a slightly aligned notion of interface automata [**dAH01a**], called I/O-transition systems, in [**BHH$^+$06**]. Since then, the quest for an appropriate theory applicable to port-based components focused on interface automata and extensions of their refinement relation based on alternating simulation [**dAH01a**]. Alternating simulation is characterised by a simulation relation that alternates between inputs and outputs. The correct refinement of a more abstract specification is required to show at least the inputs and at most the outputs as given by the specification. The freedom to add inputs, in particular in combination with the freedom to add and remove internal transitions is useful to allow implementations with additional I/O behaviour. But, in contrary, it endangers the preservation of properties that have been verified on the level of the specifications.

We considered the property output compatibility, a notion similar to "compatibility" of de Alfaro and Henzinger but also to what is called "bad activity" by Adamek and Plasil in [**AP05**]. It turned out that it is difficult to preserve compatibility if it is allowed to add internal transitions *before* an input transition of the specification is simulated in the implementation. The approach of de Alfaro and Henzinger disallows such a refinement, and we did likewise in our first approach to transfer the relation to systems with asynchronous communication in [**HJK09**]. However, we believe that refinements with additional internal transitions preceeding the simulated input transition are quite common, in particular in the setting of component-based systems where composition and abstraction are the crucial system building operators. Moreover, the preservation result for the relation developed in [**HJK09**] did not generalise for arbitrary open systems.

The solution, motivated by the aims (1) to preserve a notion of weak compatibility for both, synchronous and asynchronous communication and (2) to be transferable to open systems with asynchronous communication required the strengthening of the refinement relation with regard to the input transitions of a specification. If we do not allow to add arbitrary input transitions we are one step closer to our aims. Additionally, in order to obtain freedom with regard to preceeding internal transitions while preserving output compatibility, the preservation of these internal paths is required if it is a path to a matching input transition. Taken together we arrive at a concept of "persistent transitions" as a subset of the transitions of an IOTS that cover at least paths to inputs needed to preserve output compatibility of the specification. The resulting formalism of *input-persistent* I/O-transition

systems is a special case of modal I/O automata [**LNW07**] and the given theory might have been worked out on the basis of MIOs as considered in [**BMSH10**] for systems without buffered message exchange as well. However, we abstained from doing so, since, as argued in Sect. 1.1, the additional expressiveness seems not required in the context of output compatible environments.

In [**BMSH10**] a notion of "interface theory" is defined, providing a formal frame for a comprehensible and structured comparison of different refinement and compatibility relations for modal I/O automata. An *interface theory* is a tuple $(\mathcal{A}, \otimes, \leq, \sim)$ comprising a domain $\mathcal{A}$ of specifications, a composition operator $\otimes : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$, a reflexive and transitive refinement relation $\leq \subseteq \mathcal{A} \times \mathcal{A}$, and a symmetric compatibility relation $\sim \subseteq \mathcal{A} \times \mathcal{A}$ satisfying

(1) Compositional refinement: If $S' \leq S$ and $T' \leq T$ then $S' \otimes T' \leq S \otimes T$.
(2) Preservation of compatibility: If $S \sim T$ and $S' \leq S$ and $T' \leq T$ then $S' \sim T'$.

As argued in the following, the theory developed within this chapter defines an interface theory with asynchronous composition. PIOs with asynchronous output labelling (αPIO, Def. 4.20) provide the specification domain. Asynchronous composition is given by products of buffered PIOs (Def. 4.22), i.e., for completely buffered PIOs by $\Omega(A) \otimes \Omega(B)$; reflexivity and transitivity holds by Cor. 4.27 and we denote the corresponding composition operator by $\otimes_{\mathrm{as}}$. The refinement relation is given by blackbox refinement ($\sqsubseteq^{\mathrm{bb}}$, Def. 4.4) and the compatibility relation by asynchronous output compatibility ($\leftrightarrow_a$, Def. 4.30).

Corollary 4.54 $(\alpha\mathrm{PIO}, \otimes_{\mathrm{as}}, \sqsubseteq^{\mathrm{bb}}, \leftrightarrow_a)$ *is an interface theory.*

PROOF. (1) Compositional refinement holds by Cor. 4.29 and (2) preservation of asynchronous compatibility by Thm. 4.32. □

Finally, with the definition of greybox refinement, we distinguish a case that is important for the development of component-based systems but often neglected in the development of interface theories: the different labels of internal transitions in the refinement relation. For instance, both, interface automata with alternating simulation and modal I/O automata with weak modal refinement ignore the concrete name of an internal label. This is in fact equivalent to an approach with anonymous τ transitions. However, with such a relation we can not investigate the preservation of properties concerning the communication traces of a composed system. As to greybox refinement, we distinguish between internal labels and τ, allowing to abstract from irrelevant implementation details while, at the same time, preserving enough information to verify properties with regard to the communication traces of a system.

None of the mentioned approaches considers buffered message exchange. Thus, asynchronous compatibility was not yet considered in the context of the mentioned approaches. In Chap. 5 we will see that there is a different research community, investigating the verification of properties similar to asynchronous compatibility. However, within these approaches refinement relations are usually not considered.

CHAPTER 5

On the Verification of PIO Systems

1. Verification Based on Finite-State PIOs **76**
1.1. Compatibility, Refinement and Equivalence 76
1.2. A Criterion for Asynchronous Compatibility 77
2. Correspondence with Communicating Finite State Machines (CFSM) **81**
2.1. From I/O-Transition Systems to CFSM Systems 81
2.2. Communication Safety and Unspecified Reception 83
2.3. CFSM Systems with Local Actions 86
2.4. Queue Content Decision Diagrams (QDD) 92
2.5. QDDs and Asynchronous Communication Safety 95
2.6. On Extensions for the Verification of PIOs 97

Verification problems usually belong to one of the following categories: (1) application-specific properties such as safety properties of communication traces (2) generic properties such as reachability or compatibility (3) semantic equivalence and refinement relations. The preservation of safety properties was examined in Chap. 4 while Tab. 5.1 provides an overview of some basic problems of interest with regard to (2) and (3). In this chapter we consider foremost the verification of asynchronous communication-safety.

TABLE 5.1. Verification problems of interest

Verification	Finite	Infinite	Mixed
Reachability	$\mathscr{R}(A \otimes B)$	$\mathscr{R}(\Omega(A) \otimes \Omega(B))$	
Compatibility	$A \leftrightarrow B$	$\Omega(A) \leftrightarrow \Omega(B)$	
Refinement	$C \sqsubseteq A$	$\Omega(C) \sqsubseteq \Omega(A)$	$\Omega(C) \sqsubseteq A$
Equivalence	$A \approx A'$	$\Omega(A) \approx \Omega(A')$	$\Omega(A) \approx A'$

The problems of the first column are concerned with finite state transition systems and thus model checking techniques are readily applicable. Of course, one needs to adjust and extend existing approaches and algorithms, but there remain the fundamental principles, e.g. of reachability analysis or refinement and equivalence checking. The second column involves buffered transition systems. Since these kinds of transition systems are equipped with unbounded output queues, known problems related to the verification of infinite-state systems also apply here. As an example consider our notion of asynchronous compatibility. The definition is a requirement for the reachable states of buffered transition systems. Since composition shows, in general, again an infinite state space, the reachability problem for infinite-state systems is of interest not only for a single buffered transition system but also for their composition. For the refinement verification involving buffered transition systems, blackbox refinement between finite-state system was shown to be transferable to the buffered case (cf. Prop. 4.25). The equivalence problem, however, still exists. Moreover, there is no criteria or algorithm to decide on the refinement between a buffered and a finite transition system such as $\Omega(A) \sqsubseteq F$. The latter example directly touches the third column that shows problems where infinite and finite transition systems occur somehow “mixed”.

These kinds of verification problems are known as *regularity problems* [**Kuc99**]. Though, we are not aware of an existing approach that is applicable to the refinement verification of a buffered transition system and a non-buffered system.

The mentioned remaining open problems could be tackled by a test on the boundedness of the underlying message buffers. If the system is bounded, finite-state techniques can be applied. The approaches of Marechal et al. [**MPR04**] and Leue et al. [**LMW04**] are quite close to our formal model of asynchronously communicating systems and thus promising candidates for a transfer to a boundedness test for buffered PIO systems.

The general case, however, is that verification problems related to infinite-state systems are undecidable and one needs to find techniques with a trade-off between generality (applicable only for restricted subclasses), termination (semi-algorithms which may not halt) and exactness (over/under approximations of original system behaviour). As an example for a technique that disclaims termination, we apply a symbolic approach in the context of *Communicating Finite State Machines* (CFSM) [**vB78**]. A CFSM system consists of a set of finite state machines communicating with each other via unbounded, full-duplex, error-free FIFO-channels. With respect to analysis, early CFSM related research was usually interested in the verification of generic or application-specific properties. At the core of these problems there is the reachability problem shown to be undecidable in the seminal work of Brand and Zafiropulo [**BZ83**]. As a consequence, a number of interesting verification questions are undecidable for the general class of CFSM systems [**CF05**, Thm. 14], amongst others a communication error named *unspecified reception*. Later it will turn out that there is not only a close correspondence between buffered transition systems and CFSM systems but also in the generic properties of interest. Asynchronous comm-safety of transition systems is closely related to the notion of unspecified reception in CFSM systems.

In the remainder, we first summarise existing tools and approaches that are applicable to the verification of compatibility, refinement, and equivalence of finite-state PIOs. After that, we focus on the verification of asynchronous compatibility. First, we develop a criteria based on synchronous compatibility. Synchronous compatibility turns out to be a surprisingly weak property in comparison to asynchronous compatibility of buffered systems. Then, we elaborate the relation between systems of buffered IOTSs and CFSM systems. We consider a CFSM correspondence to asynchronous compatibility and exemplify an application of this correspondence for concrete verification. Finally, we sketch an application of a symbolic verification approach for infinite-state systems and close with remarks on required extensions for the CFSM based verification of PIOs instead of IOTSs.

1. Verification Based on Finite-State PIOs

The verification of finite-state PIOs is usually supported by existing tools or known algorithms or slight extensions hereof. We will discuss relevant references but leave the concrete implementation open. Instead we will focus on the theoretical results and partly on abstract algorithms needed for the automated verification of the problems mentioned above.

1.1. Compatibility, Refinement and Equivalence. The definition of compatibility uses the reachable state space of the product of two transition systems. The product $A \otimes B$ can be computed by standard algorithms for the computation of global state spaces. Both, explicit-state or on-the-fly algorithms could be employed. The set of reachable states $\mathscr{R}(A \otimes B)$ can be computed by a standard depth-first search in $A \otimes B$ (e.g. [**BKL08**]).

Compatibility could be verified along the computation of a transitive closure which takes the legal labels on paths towards compatible transitions into account. Legal labels are determined by the respective notion of compatibility. For weak compatibility, internal and τ-labels are legal; for ultra-weak compatibility additionally unshared output labels are allowed and finally for strong compatibility there are no legal labels at all. Verification

of output compatibility then proceeds by single-step checks between output and matching input using the corresponding transitive closure.

Besides, since PIOs are a special case of modal I/O automata, the MIO workbench, a tool for the verification of modal I/O automata [**BMSH10**] can be used to check for strong and weak compatibility between PIOs. An extension of the MIO workbench to check for ultra-weak compatibility seems to be straightforward after a discussion with the authors of [**BMSH10**]. Alternatively on may extend the approach of [**DOS09**] who implemented several kinds of compatibility checks in Maude.

Due to the correspondence between blackbox refinement and weak modal refinement as considered in [**BMSH10**] we may readily use the MIO workbench for the verification of blackbox refinement between finite state PIOs. For the case of equivalence verification, again an extension of the MIO workbench seems to be feasible. Tool-support with regard to equivalence checks should also support minimisation according to observation equivalence for PIOs similar to the LTSA tool in the context of FSP [**MKC$^+$**] and labelled transition systems.

The verification of refinement between buffered PIOs relies on Prop. 4.25 and the verification of blackbox refinement for the underlying finite-state PIOs. The refinement transfer to systems with buffered communication does not work for the mixed case, i.e. it does not hold that $C \sqsubseteq^{\mathrm{bb}} A$ implies $\Omega(C) \sqsubseteq A$ which is due to the interleaving of buffered outputs with the input transitions of the given αPIO.

1.2. A Criterion for Asynchronous Compatibility. Considering our notions of compatibility synchronous compatibility might be expected to be strong enough to entail asynchronous compatibility. In general, however, this is not the case which is due to mixed choice states, that is, due to states that show both an input and an output transition at the same time. While composition with synchronous communication works perfectly with such states, asynchronous communication introduces a buffer step whose interleaving then may lead to a deadlock within the composition of buffered transition systems. Figure 4.11 illustrates this case for the composition of two buffered PIOs.

Systems that do not show mixed choice states can be considered to be representations of single-threaded components. At any point in time, these systems either send messages, engage in local actions or receive messages and for any state with a reception there are only receptions. We call these kind of systems input-separated. A PIO A is *input separated* if for all $s \in \mathscr{R}(A)$ with $(s, l, s') \in \Delta_A$ for some $s' \in S_A$ and $l \in I_A$ it holds that $\{l' \in \mathcal{L}(L_A) \mid \exists s' \in S_A \,.\, (s, l', s') \in \Delta_A\} \subseteq I$. Fu et al. [**FBS05**] showed in the context of a formal model for web services that synchronous compatibility of input-separated systems suffices to conclude synchronizability of a given n-ary service composition where each peer is equipped with a single input queue.[1] Their proof relies on a reordering of the executions within a system of asynchronously communicating peers such that whenever a message is put into a queue it is immediately taken out of the queue. Such a reordering exists due to the input-separation of the given peers. In the following we show an analogous result for our setting of input-separated PIOs. A similar result has been proven in [**HJK09**] for closed systems of two IOTSs, strong synchronous compatibility and a slightly different notion of asynchronous compatibility.

We consider execution fragments of compositions of buffered PIOs that are initial and maximal. To simplify the presentation we focus on completely buffered PIOs. Let A be a PIO. An execution fragment $\rho \in \mathit{frag}(A)$ of A is initial and maximal if its first state is $s_{0,A}$ and it is either infinite or finite with its last state being a terminal state. With $\mathit{exec}(A)$ we denote all executions of A that are initial and maximal.

[1] It is claimed, however, that the approach can be extended to a model with one queue for each pair of communicating peers [**FBS05**, p. 1051].

Definition 5.1 (Immediate communication) *Let A and B be composable αPIOs. Let $Sys = \Omega(A) \otimes \Omega(B)$. An execution $((\sigma_{0,A}, \sigma_{0,B})\ l_1\ (\sigma_{1,A}, \sigma_{1,B})\ l_2\ \ldots) \in exec(Sys)$ is an execution with* immediate communication *if it holds that*

$$\forall l \in \mathcal{S}(A, B)\ .\ \forall i > 0\ .\ l_i = l^{\triangleright} \implies l_{i+1} = l\ .$$

Executions of buffered systems with immediate communication are related to executions of systems with synchronous communication as depicted in Fig. 5.1. The upper part of the figure shows an execution of a buffered system with one output queue. The label l is assumed to be shared between the underlying PIOs A and B, and is communicated immediately along the transition labelled l after it has been put into the queue with the transition labelled $l^{\triangleright}$. Immediate communication implies, due to the FIFO ordering of queues, that the particular queue was empty before the enqueue action.

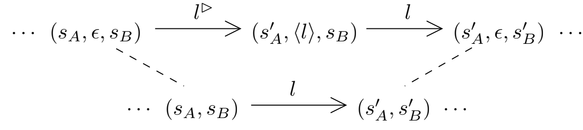


FIGURE 5.1. Immediate communication and synchronous executions

The set of persistent transitions Π of PIOs does not play a decisive role for the relation between states indicated in Fig. 5.1. Assume l is an output of the underlying PIO A which is not persistent. Then all transitions are neither persistent and the relation holds as is; if the output l is persistent then the enqueue action is persistent while persistence of the communication depends on the input transition of B. If $(s_B, l, s'_B) \in \Pi_B$ then the communication is persistent; if $(s_B, l, s'_B) \notin \Pi_B$ then the communication is neither persistent. In both cases, however, the relation between the states in Fig. 5.1 is not touched.

The basic idea of [**FBS05**] is to show that for any execution of a buffered system composed of compatible input-separated peers, there exists an equivalent execution with immediate communication. Equivalence is based on communication traces of the system. Let A and B be composable αPIOs. The *communication trace* $\mathscr{C}$ of an execution $\rho \in exec(\Omega(A) \otimes \Omega(B))$ is defined inductively: If $\rho = (\sigma_{0,A}, \sigma_{0,B})$ then $\mathscr{C}(\rho) = \epsilon$; if $\rho = \rho' \cdot (\sigma_A, \sigma_B) l (\sigma'_A, \sigma'_B)$ then $\mathscr{C}(\rho) = \mathscr{C}(\rho') \cdot l$ if $l \in \bigcup L_A \cup \bigcup L_B$ and $\mathscr{C}(\rho) = \mathscr{C}(\rho')$ otherwise. Besides input-separation we need to assume that A and B do not diverge on output or internal actions. Otherwise there are infinite paths in the buffered system that can not be related to synchronous executions. Such a relation, however, is fundamental for the proof of the succeeding proposition. An αPIO A *diverges autonomously* if there exists an infinite execution $s_0 l_1 s_1 l_2 \ldots \in exec(A)$ such that $l_i \in O_A \cup \mathcal{T}(A)$ for all $i \geq k$ for some $k \geq 1$.

Proposition 5.2 *Let A and B be composable αPIOs such that (A, B) is a closed system, and both are input-separated and do not diverge autonomously. If $A \leftrightarrow_w B$ then for all $\rho \in exec(\Omega(A) \otimes \Omega(B))$ there exists $\rho' \in exec(\Omega(A) \otimes \Omega(B))$ such that $\mathscr{C}(\rho) = \mathscr{C}(\rho')$ and ρ' is an execution with immediate communication.*

PROOF. By induction on the number of labels in the communication trace for ρ. We prove that for all $0 \leq n \leq |\mathscr{C}(\rho)|$ there exists ρ' with $\mathscr{C}(\rho) = \mathscr{C}(\rho')$ and the first n enqueue actions in ρ' are immediately communicated. Let $\mathscr{C}(\rho) = l_1 l_2 \ldots l_{|\mathscr{C}(\rho)|-1}$.

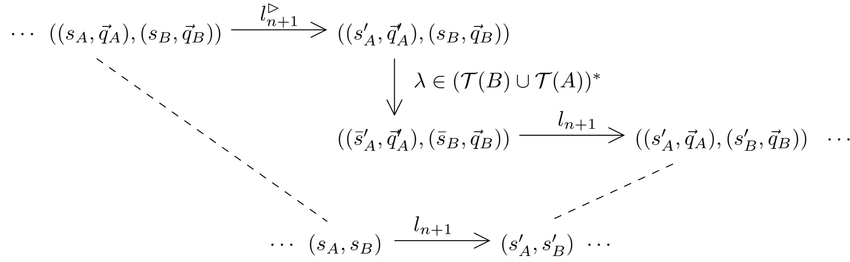
For the base case $n = 0$ we have $\mathscr{C}(\rho) = \epsilon$ and the claim holds trivially. Induction hypothesis (IH): For all $1 \leq n \leq |\mathscr{C}(\rho)|$ there exists ρ' such that $\mathscr{C}(\rho) = \mathscr{C}(\rho')$ and the first n enqueue actions in ρ' are immediately communicated.

Induction step $(n+1)$. Let $Sys = \Omega(A) \otimes \Omega(B)$. We consider an output of $\Omega(A)$, the case for $\Omega(B)$ is symmetric. Let $l_{n+1} \in O_A$ and assume

(*) $\qquad ((\sigma_{n,A}, \sigma_{n,B}), l^{\triangleright}_{n+1}, (\bar{\sigma}_{n,A}, \sigma_{n,B})) \in \Delta_{Sys}$ is the $n+1$ step in ρ.

First we show that the first non-internal action of $\Omega(B)$ in Sys is the reception (i.e. input) of l_{n+1}. With (IH) it holds for ρ' that all enqueue actions labelled $l^{\triangleright}$ in ρ' are immediately followed by a corresponding communication transition labelled l. Let $\sigma_{n,A} = (s_A, \vec{q}_A)$, $\sigma_{n,B} = (s_B, \vec{q}_B)$ and denote the output queue of A for the shared labels $O_A \cap I_B$ in $\vec{q}_A$ by $q_A^{\triangleright B}$. Since all queues are initially empty, it follows that $q_A^{\triangleright B} = \epsilon$. The execution ρ' is used for an inductive construction of the fragment $((s_{0,A}, s_{0,B})l_1 \cdots l_n(s_A, s_B)) \in frag(A \otimes B)$, hence $(\sigma_{n,A}, \sigma_{n,B})$ and (s_A, s_B) are in relation.

With (*) we have $((s_A, \vec{q}_A), l^{\triangleright}_{n+1}, (s'_A, \vec{q}'_A)) \in \Delta_{\Omega(A)}$ and thus $(s_A, l_{n+1}, s'_A) \in \Delta_A$. Now, by $A \leftrightarrow_w B$, there exists $s'_B \in S_B$ such that $(s_B, l_{n+1}, s'_B) \in \Pi_B^{\mathcal{T}(B)}$. Since $A \leftrightarrow_w B$ is a requirement for $\mathscr{R}(A \otimes B)$, it holds for all $\bar{s}_B \in S_B$ with $(s_B, \bar{s}_B) \in \Pi_B^{\mathcal{T}(B)}$ and $(\bar{s}_B, l_{n+1}, s'_B) \in \Pi_B$ that $\bar{s}_B$ has an outgoing input transition and since B is input-separated, it follows that $\bar{s}_B$ shows only outgoing input transitions. The state $\bar{s}_B$ is reached in Π_B by internal transitions with labels from $\mathcal{T}(B)$ only. Hence, if $(s_B, \vec{q}_B) \in \mathscr{R}(\Omega(B))$ then $(\bar{s}_B, \vec{q}_B) \in \mathscr{R}(\Omega(B))$ such that $(\bar{s}_B, \vec{q}_B)$ shows outgoing input transitions only. Since $q_A^{\triangleright B} = \epsilon$ it follows that the execution fragment ending in (*) above can be extended and set into relation with a synchronous execution as follows:



Consider the state $((s'_A, \vec{q}'_A), (s_B, \vec{q}_B))$ after the enqueue action. Since (A, B) is a closed system there can not be any outgoing non-shared transitions, interleaved in the composition of $\Omega(A)$ and $\Omega(B)$. Thus, with respect to actions of $\Omega(B)$ the only continuation other than communicating l_{n+1} are labelled by $\mathcal{T}(B)$. Since A is input-separated the only continuation concerning actions of $\Omega(A)$ are local actions labelled by $\mathcal{T}(A)$ and further enqueue actions $l^{\triangleright}$ with $l \in O_A$.

Since A and B are not autonomously diverging, local actions do not affect the communication traces of $\Omega(A) \otimes \Omega(B)$ and can be ignored. Consider state $((\bar{s}'_A, \vec{q}'_A), (\bar{s}_B, \vec{q}_B))$ after the sequence of transitions with labels in λ. If $\Omega(A)$ does not show further enqueue actions we are done, since $l^{\triangleright}_{n+1}$ and the labels in λ are local actions in $\Omega(A) \otimes \Omega(B)$, we may safely rearrange the ordering such that $l^{\triangleright}_{n+1}$ is immediately followed by the communication transition labelled l_{n+1} and obtain an execution ρ' such that $\mathscr{C}(\rho) = \mathscr{C}(\rho')$ and the first $n + 1$ enqueue actions are immediately followed by a corresponding communication and the queue $q_A^{\triangleright B}$ is empty again.

If $((\bar{s}'_A, \vec{q}'_A), (\bar{s}_B, \vec{q}_B))$ continues with enqueue actions of $\Omega(A)$ then, since A is not diverging autonomously, there exists a state $(s''_A, \vec{q}''_A) \in S_{\Omega(A)}$ that is either terminal or a state with outgoing input-transition. The state $((s''_A, \vec{q}''_A), (\bar{s}_B, \vec{q}_B))$ is reached after a sequence of transitions labelled by $\lambda_A^{\triangleright} \in (O_A \cap I_B)^*$. Since A is input-separated and B is still awaiting an input of l_{n+1}, the only possible continuation from $((s''_A, \vec{q}''_A), (\bar{s}_B, \vec{q}_B))$ is the communication of l_{n+1}. The resulting execution sequence with labels $\lambda_A^{\triangleright} \cdot l_{n+1}$ can be reordered to an execution with labels $l_{n+1} \cdot \lambda_A^{\triangleright}$ without effect on the communication trace of the given system and again in relation to a synchronous execution as indicated in the figure above. The same argumentation holds until a state $((s''_A, \vec{q}_A), (s''_B, \vec{q}_B))$ with $q_A^{\triangleright B} = \epsilon$ is reached. □

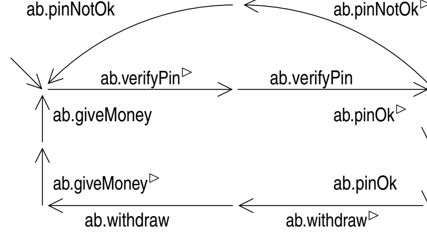


FIGURE 5.2. Assembly behaviour of the Bank/Atm system of Fig. 1.3

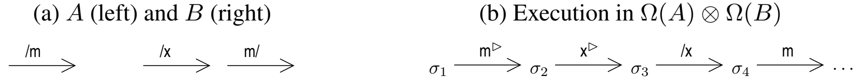


FIGURE 5.3. Ultra-weak output compatibility and buffered executions

Theorem 5.3 (Weakly asynchronous output compatibility) *Let A and B be composable αPIOs such that (A, B) is a closed system, and both are input-separated and do not diverge autonomously. If $A \leftrightarrow_w B$ then $\Omega(A) \leftrightarrow_w \Omega(B)$.*

PROOF SKETCH. Let $\rho \in exec(\Omega(A) \otimes \Omega(B))$. By Prop. 5.2 we obtain from ρ an equivalent execution ρ' with immediate communication. With ρ' we relate the states within $\Omega(A) \otimes \Omega(B)$ to the states within a synchronous execution in $A \otimes B$ as depicted in Fig. 5.1 and the proof of Prop. 5.2. By this means we obtain for any local output in $\Omega(A) \otimes \Omega(B)$ a weakly matching input using $A \leftrightarrow_w B$ and hence $\Omega(A) \leftrightarrow_w \Omega(B)$. □

Example 5.4 *The composition of the transition systems for the behaviour of the Bank and Atm components as introduced in Fig. 1.3 of Chap. 1 results in a closed system as indicated by Fig. 5.2. Due to the application of an asynchronous connector in the static structure specification, a buffering behaviour of the connector ab is involved. Labels of the form ab.$m^{\triangleright}$ represent the action of sending a message m on the connector ab which means to put the message into a message buffer managed by the connector. Labels of the form ab.m represent the action of taking a message out of the buffer. The transition systems in Fig. 1.3 are both input-separated and always eventually receiving. Moreover they are weakly output compatible, since any shared output is immediately synchronised by an input of the particular communication partner. Thus Thm. 5.3 is applicable and it follows that the behaviours are weakly asynchronously compatible.*

Since [**FBS05**] considered n-ary composition of web service behaviours, we believe that a corresponding generalisation of Thm. 5.3 to conclude weak asynchronous comm-safety of n-ary closed systems is feasible. Ultra-weak output compatibility makes a difference only within open systems. For the case of open systems, already the case of two communicating transition systems imposes the need for an aligned definition of "immediate communication". Consider for instance the transition systems in Fig. 5.3. Assume $\mathsf{x} \notin \mathcal{S}(A, B)$, then the buffered system resulting from the composition of $\Omega(A)$ and $\Omega(B)$ shows an execution $\sigma_1\ \mathsf{m}^{\triangleright}\ \sigma_2\ \mathsf{x}^{\triangleright}\ \sigma_3\ \mathsf{x}\ \sigma_4\ \mathsf{m} \ldots$ that can not be reordered to an execution with immediate receives without affecting its communication trace. Note that for weak output compatibility this case could not arise since B only engaged in internal actions before the matching input is reached. In case of ultra-weak output compatibility additionally non-shared outputs may occur on these paths as illustrated by the given example.

2. Correspondence with Communicating Finite State Machines (CFSM)

Our model of asynchronous communication is closely related to systems of communicating finite state machines. CFSM systems are usually considered without internal actions, that is, the only action are either enqueue or dequeue actions and internal or τ moves are not taken into account. Within this section we first provide a translation for a corresponding class of IOTSs. Such a translation is useful not only for a deeper understanding of the interrelation between our model of asynchronously communicating components and CFSM systems but also to make a large number of interesting research results, e.g. boundedness tests [**LMW04**] or criteria for the decidability of reachability problems [**CF05**], applicable to the verification of buffered IOTSs. In this context we investigate a correspondence between asynchronous compatibility and unspecified reception, a well-known communication error in CFSM systems, showing that freedom of unspecified reception is not sufficient to conclude asynchronous compatibility.

The work of Boigelot et al. [**BG99, BGWW97**] on the symbolic verification of systems with unbounded FIFO queues relies on a definition of CFSM systems with local actions. We provide again a translation for a corresponding class of IOTSs which makes their approach applicable to the verification of IOTSs with internal moves. After that we sketch an application to the verification of asynchronous compatibility and finally we discuss how to extend the approach to cope with PIOs instead of IOTSs.

2.1. From I/O-Transition Systems to CFSM Systems. A list of general CFSM verification problems can be found, for instance, in [**CF05**] which was also used as a starting point for the succeeding CFSM related definitions.

Definition 5.5 (CFSM cf. [**CF05**]) *A* communicating finite state machine (CFSM) *is a tuple $M = (Q, q_0, \Sigma, \delta)$ where Q is a finite set of states, $q_0 \in Q$ an initial state, Σ an alphabet and $\delta \subseteq Q \times (\mathbb{N} \times \{!, ?\} \times \Sigma) \times Q)$ a finite set of transitions.*

We write $(q, j!l, q') \in \delta$ instead of $(q, (j, !, l), q') \in \delta$ and analogously for labels involving $?$. Moreover we define $\Sigma^{\text{snd}} = \{l \in \Sigma \mid \exists q, q' \in Q \,.\, \exists j \in \mathbb{N} \,.\, (q, j!l, q') \in \delta\}$ and analogously Σ^{rcv} for $(q, j?l, q') \in \delta$.

The transition labels of CFMSs comprise a unique integer that identifies the channel to be used for sending (!) or receiving receiving (?) messages defined by alphabet Σ. The given CFSMs correspond to finite IOTSs which communicate completely asynchronous and do neither show internal nor τ transitions. The class of *completely asynchronous pure IOTSs* consists of finite I/O-transition systems $A = (L, S, s_0, \Delta)$ with partitioned I/O-labelling L such that $((L, S, s_0, \Delta, \Pi), O^{\triangleright})$ is an αPIO with $\bigcup L = \bigcup O^{\triangleright}$ and $\Delta = \Pi$.

Definition 5.6 (CFSM translation) *Let A be a completely asynchronous pure IOTS. The* cfsm-translation from A to a CFSM *is given by $cfsm(A) = (Q, q_0, \Sigma, \delta)$ such that $Q = S_A$, $q_0 = s_{0,A}$, $\Sigma = \bigcup L_A$ and for $L = ((I_1, O_1, \ldots, I_n, O_n), T)$ for all $1 \leq j \leq n$:*

$$(s, j!l, s') \in \delta \iff \exists l \in O_j \,.\, (s, l, s') \in \Delta_A,$$
$$(s, j?l, s') \in \delta \iff \exists l \in I_j \,.\, (s, l, s') \in \Delta_A.$$

Note that CFSMs do not distinguish send and receive label within Σ and therefore the mapping is, in general, not reversible. However, often there is no need to consider the complete alphabet and one may reconstruct at least the subset $\Sigma^{\text{snd}} \cup \Sigma^{\text{rcv}} \subseteq \Sigma$ used in δ.

Communicating finite state machines together with a set of FIFO-buffered communication channels constitute a formal framework for the analysis of communication protocols. Usually it is assumed that between each pair of CFSMs there are two channels, thus allowing for separated unidirectional buffered communication. There are different classes of CFSM systems along distinguished properties of their channels. Of course, first of all, the channels might be bounded or unbounded. Second the channels may "loose" messages

during transmission from a sender to a receiving CFSM. These kind of systems are a possible means to model communication along unreliable transmission mediums. In contrast "perfect channels" do not loose any message during transmit. Moreover, the classification might also distinguish along specific properties such as a "well-ordering" of messages in order to identify particular restricted system classes for the study of decidability questions. A corresponding list comprising also the mentioned classes can be found in [**CF05**, Sect. 1.1]. In the following we consider CFSM systems with unbounded perfect FIFO-channels.

Definition 5.7 (CFSM System, cf. [**CF05**]) *A* communicating system *is a tuple* $Sys = (M_1, \ldots, M_n)$ *of CFSMs* $M_i = (Q_i, q_{0,i}, \Sigma_i, \delta_i)$ *with pairwise disjoint alphabets* Σ_i*.*

Let p be the number of channels in Sys[2]. The *global state space* (set of configurations) Γ_{Sys} of Sys is defined by $\Gamma_{Sys} = Q_1 \times \ldots \times Q_n \times (\Sigma^*)^p$, where $\Sigma = \bigcup_i \Sigma_i$. A state $\gamma \in \Gamma_{Sys}$ is called a *configuration* of Sys. The *initial configuration* is given by $\gamma_{0,Sys} = ((q_{0,1}, \ldots q_{0,n}, \epsilon_1, \ldots \epsilon_p))$. The operational semantics of a communicating system Sys induces a *transition relation* $\rightarrow_{Sys} \subseteq \Gamma_{Sys} \times \Gamma_{Sys}$, where

$$((q_1, \ldots q_n, x_1, \ldots x_p), (q'_1, \ldots q'_n, x'_1, \ldots x'_p)) \in \rightarrow_{Sys},$$

if there exists $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, p\}$ and $l \in \Sigma$ such that one of the following holds:

$$((q_i, j!l, q'_i) \in \delta_i \wedge q'_k = q_k \text{ for all } k \neq i) \wedge (x'_j = x_j l \text{ and } x'_h = x_h \text{ for all } h \neq j),$$
$$((q_i, j?l, q'_i) \in \delta_i \wedge q'_k = q_k \text{ for all } k \neq i) \wedge (l x'_j = x_j \text{ and } x'_h = x_h \text{ for all } h \neq j).$$

For $(\gamma, \gamma') \in \rightarrow$ we write $\gamma \rightarrow \gamma$, or redundantly, $\gamma \xrightarrow{ls} \gamma'$ for $ls \in \{(q_i, j!l, q'_i), (q_i, j?l, q'_i)\}$; we write $label(ls)$ to extract the label $j!l$, $j?l$ respectively from the given local step.

The notation $\gamma \xrightarrow{ls} \gamma'$ shows the local step $ls \in \{(q_i, j!l, q'_i), (q_i, j?l, q'_i)\}$ which triggered the particular global transition. Therefore, a communicating system can be understood as a labelled transition system. The formal underpinning of this view in [**CF05**] is the reachability set and reachability graph of Sys. The *reachability set* $\mathscr{R}(Sys)$ of a communicating system Sys is defined inductively: $\gamma_{0,Sys} \in \mathscr{R}(Sys)$; and if $\gamma \in \mathscr{R}(Sys)$ and there is $\gamma' \in \Gamma_{Sys}$ such that $\gamma \rightarrow_{Sys} \gamma'$, then $\gamma' \in \mathscr{R}(Sys)$. The set of states $\mathscr{R}(\gamma)$, reachable from a given $\gamma \in \mathscr{R}(Sys)$ is defined analogously using the reachability set of Sys. The reachability graph is a labelled graph whose nodes are given by $\mathscr{R}(Sys)$ and whose edges correspond to global transitions according to the notation $\gamma \xrightarrow{ls}_{Sys} \gamma'$. Thus the reachability graph represents a transition system similar to the composition of buffered I/O-transition systems. In the following we define a translation $iots$ from Sys to a (composed) buffered I/O-transition system and after that discuss the correspondence between products of buffered IOTSs and the $iots$-translation of the communicating system which consists of $cfsm$-translations of the given IOTSs.

The iots-translation of a communicating system yields a completely asynchronous pure IOTS as follows. Denote the selection of projected channel alphabets by $(\Sigma^*)^p \downarrow_{\Sigma_i}$, which selects the $h \leq p$ projections $\Sigma \restriction_{i,1} \times \cdots \times \Sigma \restriction_{i,h}$ of Σ to the ith alphabet Σ_i such that $\Sigma \restriction_{i,j} \cap \Sigma_i \neq \emptyset$ for all $1 \leq j \leq h$ (the projections $\Sigma \restriction_{i,j}$ yield the alphabet of the jth queue of the ith transition system, i.e. $\Sigma \restriction_{i,1} \times \cdots \times \Sigma \restriction_{i,h}$ is a vector of all queues where symbols from Σ_i are involved).

Definition 5.8 (Sys IOTS) *Let* $Sys = (M_1, \ldots, M_n)$ *be a communicating system and let* $z_i = |(\Sigma^*)^p \downarrow_{\Sigma_i^{\mathrm{snd}}}|$ *for* $1 \leq i \leq n$*. The* iots-translation *of* Sys *is a completely asynchronous pure IOTS given by* $iots(Sys) = (L, S, s_0, \Delta)$*, where*

[2]The number of edges in a complete graph is $n(n-1)/2$, thus for the case of two channels between any pair of CFSMs we would have $p = n(n-1)$ queues.

1. $L = (I, O, T)$*, where* $I = \emptyset$*,* $O = \emptyset$ *and*

$$T = \bigcup_{i=1}^{n}\{l^{\rhd} \mid l \in \Sigma_i^{\mathrm{snd}}\} \cup \bigcup_{i=1}^{n}\{l \mid l \in \Sigma_i^{\mathrm{rcv}}\} .$$

2. S *and* s_0 *is obtained by a reordering based on* $(\Sigma^*)^p\downarrow_{\Sigma_i}$*, such that*

$$(Q_1 \times \ldots \times Q_n \times (\Sigma^*)^p) \mapsto (Q_1 \times (\Sigma^*)^p\downarrow_{\Sigma_1^{\mathrm{snd}}} \times \cdots \times Q_n \times (\Sigma^*)^p\downarrow_{\Sigma_n^{\mathrm{snd}}}),$$
$$(q_{0,1}, \ldots, q_{0,n}, \epsilon_1, \ldots, \epsilon_p) \mapsto (q_{0,1}, (\epsilon)^{z_1}, \ldots, q_{0,n}, (\epsilon)^{z_n}).$$

3. $\Delta \subseteq S \times \bigcup L \times S$ *is the smallest relation such that*

(i) if $(\vec{q}, \vec{x}) \xrightarrow{(q_i, j?l, q_i')}_{Sys} (\vec{q}', \vec{x}')$ *then* $(f(\vec{q}, \vec{x}), l, f(\vec{q}', \vec{x}')) \in \Delta$*, where*

$$(\vec{q}, \vec{x}) = ((q_1, \ldots, q_i, \ldots, q_n), (x_1, \ldots, lx_j, \ldots, x_p)),$$
$$(\vec{q}', \vec{x}') = ((q_1, \ldots, q_i', \ldots, q_n), (x_1, \ldots, x_j, \ldots, x_p)),$$
$$f(\vec{q}, \vec{x}) = ((q_1, \vec{x}_1), \ldots, (q_i, \vec{x}_i), \ldots, (q_k, (x_{k,1}, \ldots, lx_{k,j}, \ldots, x_{k,z_k})), \ldots, (q_n, \vec{x}_n)),$$
$$f(\vec{q}', \vec{x}') = ((q_1, \vec{x}_1), \ldots, (q_i', \vec{x}_i), \ldots, (q_k, (x_{k,1}, \ldots, x_{k,j}, \ldots, x_{k,z_k})), \ldots, (q_n, \vec{x}_n)).$$

(ii) if $(\vec{q}, \vec{x}) \xrightarrow{(q_i, j!l, q_i')}_{Sys} (\vec{q}', \vec{x}')$ *then* $(f(\vec{q}, \vec{x}), l^{\rhd}, f(\vec{q}', \vec{x}')) \in \Delta$*, where*

$$(\vec{q}, \vec{x}) = ((q_1, \ldots, q_i, \ldots, q_n), (x_1, \ldots, x_j, \ldots, x_p)),$$
$$(\vec{q}', \vec{x}') = ((q_1, \ldots, q_i', \ldots, q_n), (x_1, \ldots, x_j l, \ldots, x_p)),$$
$$f(\vec{q}, \vec{x}) = ((q_1, \vec{x}_1), \ldots, (q_i, (x_{i,1}, \ldots, x_{i,j}, \ldots, x_{i,z_i})), \ldots, (q_n, \vec{x}_n)),$$
$$f(\vec{q}', \vec{x}') = ((q_1, \vec{x}_1), \ldots, (q_i', (x_{i,1}, \ldots, x_{i,j}l, \ldots, x_{i,z_i})), \ldots, (q_n, \vec{x}_n)).$$

The translation $cfsm(A)$ of a single IOTS A and the IOTS-view $iots(Sys)$ of a communicating system Sys allows to draw a correspondence between CFSM systems and buffered I/O-transition systems. Since CFSM systems are closed systems we need an additional assumption on the closedness of the given IOTSs. Pairwise composable IOTSs $(A_1, \ldots, A_n)$ are a *closed system*, if $I_{\otimes_i A_i} = O_{\otimes_i A_i} = \emptyset$, i.e. if the remaining sets of input and output labels are empty. We state the following without formal proof. Moreover, the translation of labels in the IOTS-view takes only those labels into account that actually have been used within the local transition relations δ_i, that is, labels from $\Sigma^{\mathrm{snd}} \cup \Sigma^{\mathrm{rcv}}$. This syntactical difference from the alphabets of the underlying IOTSs could be handled by a corresponding alphabet extension. To simplify the presentation we assume that all labels have actually been used in transitions and call such an IOTS *without unused labels*.

Proposition 5.9 (Correspondence) *Let* $(A_1, \ldots, A_n)$ *be a closed system of pairwise composable completely asynchronous pure IOTSs without unused labels. Then* $(cfsm(A_1), \ldots, cfsm(A_n)) = \bigotimes_i \Omega(A_i)$*.* □

2.2. Communication Safety and Unspecified Reception. We consider the notion of unspecified reception configurations. For the precise definition in the sense of [**CF05**] and supported by an earlier understanding in [**GMY84**], we need a notion of "receiving state". Even though not necessary for the definition of unspecified reception we consider also "mixed states" as a prerequisite for later discussions. A state $q \in Q$ of a CFSM $M = (Q, q_0, \Sigma, \delta)$ is a *receiving state*, if all of its outgoing transitions are receiving transitions; a *mixed state* shows both, a sending and a receiving transition (at least one of each). We use these notions for IOTSs analogously. The following definition extends [**CF05**, Def. 12] for unspecified receptions.

Definition 5.10 (Ineffective reception) *Let* $Sys = (M_1, \ldots, M_n)$ *be a communicating system. A configuration* $\gamma = (q_1, \ldots, q_n, x_1, \ldots, x_p) \in \Gamma_{Sys}$ *is an* ineffective reception configuration*, if there exists* $i \in \{1, \ldots, n\}$ *such that* q_i *is a receiving or a mixed state and*

$$\forall l \in \Sigma_i \,.\, \forall q_i' \in Q_i \,.\, (q_i, j?l, q_i') \in \delta_i \implies (|x_j| \geq 1 \wedge x_j \neq l\,\Sigma^*).$$

If q_i is a receiving state, then γ is an unspecified reception configuration*, if it is a mixed state we call γ an* ignored reception configuration*. The system Sys is free from unspecified (ignored) receptions, if γ is not an unspecified (ignored) reception configuration for all $\gamma \in \mathscr{R}^{\text{snd}}(Sys)$.*

In an unspecified reception configuration there exists a machine M_i which is stuck at a receiving state. Being stuck means here, that M_i can not proceed with any of its outgoing receive transitions $(q_i, j?l, q_i')$ since the topmost message of the jth queue is different from l. This kind of error state reminds of our original motivation for the introduction of asynchronous compatibility for transition systems with buffered communication. Given some local output, we aimed for the unbuffered, synchronous case to guarantee the existence of a matching input within a composed system such that messages sent do not "disappear" due to a missing reception (synchronisation). A natural transfer of this property to the buffered case resulted in our definition of asynchronous compatibility for buffered transition systems. The requirement "not to disappear" was then understood as "enqueued messages are eventually dequeued" that is eventually processed. And indeed, we have the following implication.

Proposition 5.11 *Let $(A_1, \dots, A_n)$ be a closed system of pairwise composable completely asynchronous pure IOTSs. If $(\Omega(A_1), \dots, \Omega(A_n))$ is ultra-weakly comm-safe, then $(cfsm(A_1), \dots, cfsm(A_n))$ is free from unspecified receptions.*

PROOF. The proposition is a special case of Prop. 5.16 along Cor. 5.15. □

As witnessed by the translation of the transition systems in Fig. 5.4 and argued in the following example the reverse direction is not true.

Example 5.12 (Unspecified reception) *Figure 5.4 shows two pairs of I/O-transition systems whose CFSM-translation yield a communication system which is free of unspecified receptions. The IOTSs on the lefthand side do not show receiving states at all, hence there is no unspecified reception. In contrast, the IOTSs on the righthand side do show receiving states but there is also the chance to enter a path (along input x) where the receiving state is missed. The input transition moves to a state without any outgoing transition, and since the initial state is neither a receiving state we have again freedom from unspecified reception.*

FIGURE 5.4. Free from unspecified reception

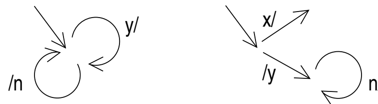


FIGURE 5.5. Receiving states

Both pairs of transition systems are not asynchronously output compatible due to missing receive transitions. Figure 5.5 shows a revision of the righthand pair aimed to repair that defect, but again the transition systems are not asynchronously compatible. The incompatibility reveals a fundamental difference between unspecified reception and asynchronous compatibility.

For a first illustration observe that an extension of the machine sending n *by an arbitrary receive transition different from receiving* y *would also result in a CFSM system which is free from unspecified reception. The latter may come as a surprise but indeed the*

requirement for unspecified reception is for receiving states only and the given transition system shows a mixed state, not a receiving state. Thus we would have an ignored reception configuration which is a communication error different from what is commonly understood as unspecified reception. From a conceptual point of view this means that unspecified reception is supposed to catch up situations where a machine is definitely stucked which is obviously not the case for the given mixed-choice system since it is always able to send out messages n *without ever using its receive transition. This is quite some difference to our definition of asynchronous output compatibility with a requirement for all reachable states of a given composition. With the definition of ignored reception configurations we are one step closer. The discussed system with an input different from* y *"ignores" the necessity to receive* y *and continues to send* n *instead.*

Still the transition systems in Fig. 5.5 are not asynchronously compatible due to the input transition x *which globally allows to reach a state where the input* n *is not received. Adding a loop after* x *with input* n *solves this problem and the resulting system is indeed asynchronously compatible.*

In both given examples there are mixed states which are not asynchronously compatible but skipped over by the definition of unspecified reception due to their property of not being a receiving state. A third problem arises from pure send states such as the one given on the lefthand side of Fig. 5.4. If there is neither a receiving nor a mixed state none of the mentioned requirements applies. In order to close this last gap we expect a transition system to provide at last any receive transition in case a state with a non-empty queue is entered. More precisely, let $M = (Q, q_0, \Sigma, \delta)$ be a CFSM and let $q \in Q$. The *states reachable from q via* snd*-transitions* are given inductively by $q \in \mathscr{R}^{\mathrm{snd}}(q)$; and if $q' \in \mathscr{R}^{\mathrm{snd}}(q)$ and there is $q'' \in Q$ such that $(q', l, q'') \in \delta$ and $l \in \Sigma_M^{\mathrm{snd}}$ then $q'' \in \mathscr{R}^{\mathrm{snd}}(q)$. Moreover, *$q$ is a receiving or mixed state for $j \in \mathbb{N}$* if there exists $l \in \Sigma_M^{\mathrm{rcv}}$, $q' \in Q$ such that $(q, j?l, q') \in \delta$.

Definition 5.13 (Eventually receiving) *Let $Sys = (M_1, \ldots, M_n)$ be a communicating system. A configuration $\gamma = (q_1, \ldots, q_n, x_1, \ldots, x_p) \in \Gamma_{Sys}$ is* eventually receiving*, if for all $j \in \{1, \ldots, p\}$ there exists $i \in \{1, \ldots, n\}$ such that the following holds:*

$$x_j \neq \epsilon \implies (\exists (q'_1, \ldots, q'_n, x'_1, \ldots, x'_p) \in \mathscr{R}(\gamma) \,.\, (q'_i \in \mathscr{R}^{\mathrm{snd}}(q_i) \wedge q'_i \text{ is a receiving or a mixed state for } j)) \,.$$

Definition 5.14 (Asynchronously responsive) *A communicating system Sys is* asynchronously responsive*, if for all $\gamma \in \mathscr{R}(Sys)$ it holds that γ is eventually receiving and is neither an unspecified nor an ignored reception configuration.*

Corollary 5.15 *If a communicating system Sys is asynchronously responsive, then it is free from unspecified reception.* □

In the following we sketch a proof for the correspondence of ultra-weak comm-safety and asynchronous responsiveness. By this means comm-safety of buffered IOTSs can be verified along tools for the verification of CFSM systems and, in contrary, knowing ultra-weak comm-safety for a given system of IOTSs can be used to conclude asynchronous responsiveness, and hence unspecified reception, of the corresponding CFSM system. Note that we consider closed systems and hence ultra-weak comm-safety coincides with weak comm-safety. An example for the verification of asynchronous responsiveness is given in Ex. 5.29 for a simple Bank/ATM system.

Proposition 5.16 (Asynchronously responsive) *Let $(A_1, \ldots, A_n)$ be a closed system of pairwise composable completely asynchronous pure IOTSs. $(\Omega(A_1), \ldots, \Omega(A_n))$ is ultra-weakly comm-safe iff the communicating system $(cfsm(A_1), \ldots, cfsm(A_n))$ is asynchronously responsive.*

PROOF SKETCH. "$\Rightarrow$" Let $\Omega(A_i)$ be the buffered system with local output $^{\triangleright}$ in the given reachable global state. Then one of the queues is not empty. By ultra-weak comm-safety there is a state σ globally reachable such that l becomes the topmost message of this queue. Since ultra-weak comm-safety requires an autonomously reachable matching input transition and autonomously in a closed system without τ and internal transitions means reachable by local snd-transitions we obtain the property eventual receiving for σ. Thus there exists a snd-reachable σ' which provides the matching input transition. If σ' is a mixed state, then by the requirement of ultra-weak comm-safety, it holds that σ' is not an ignored reception; also, if σ' is a receiving state it holds that σ' is not an unspecified reception. Since ultra-weak comm-safety is a requirement for the complete reachable state space this means that the communicating system obtained from the cfsm-translation of the given IOTSs is asynchronously responsive.

"$\Leftarrow$" Asynchronous responsiveness is a requirement for the complete state space, hence the argumentation proceeds analogously. Due to the property eventually receiving we find receiving or mixed states γ', reachable from some given queue state with message l being topmost. By definition, reachability is via snd-transitions and thus γ' is globally reachable. In case of a receive state we have by the assumption of unspecified reception that the input transitions matches; in case of a mixed state, we use the assumption of ignored receptions to obtain the matching input. Therefore we have a witness for ultra-weak comm-safety of γ' and the claim follows by asynchronous responsiveness being a complete state space requirement. □

By Prop. 5.16 one may apply CFSM verification techniques to show asynchronous responsiveness in order to verify asynchronous compatibility of the considered class of IOTSs. Now the complexity lies in the three conditions for asynchronous responsiveness: without unspecified reception, without ignored reception and eventually receiving. Since the notion of unspecified reception is a well-known topic for CFSM verification it would be interesting to get a similar understanding of CFSM systems that are eventually receiving and without ignored receptions. The progress requirements of Gouda et al. [**GMY84**] seems to come closest. Here, a state is called *non-progress* state if and only if it is an unspecified reception, a deadlock or an overflow state. Obviously, asynchronous responsiveness does not include deadlock freedom but systems without overflow states seem to be quite close to the systems that are eventually receiving and without ignored reception. A fully detailed and formal account on these issues could be an interesting issue for future work.

2.3. CFSM Systems with Local Actions. Boigelot et.al. [**BG99, BGWW97**] developed and applied a special kind of finite state machines, *Queue content Decision Diagrams (QDD)*, for a symbolic representation of the queue contents of CFSM systems. By this means, potentially infinite-state systems can be represented in a finite way, paving the way for an application of model-checking techniques for the verification systems with unbounded FIFO queues. Their definition of CFSM systems include local actions, that is, actions local to a CFSM without an effect on the queues of the given system of CFSMs. In the following we describe a translation between a corresponding class of IOTSs and CFSM systems with local actions. After that, we introduce QDDs and investigate their applicability to the verification of asynchronous compatibility.

For a better understanding of the relationship and translation between IOTSs and CFSMs we provide an explicit account of what we call CFSM with local actions.

Definition 5.17 (Extended CFSM) *A* CFSM with local actions *(extended CFSM) is a tuple $M = (Q, q_0, \Sigma^e, \delta^e)$ where Q is a finite set of states, $q_0 \in Q$ an initial state, $\Sigma^e = \Sigma^c \cup \Sigma^a$ alphabets for communication and local actions, and $\delta \subseteq Q \times (\Sigma^a \cup (\mathbb{N} \times \{!, ?\} \times \Sigma^c)) \times Q$ a finite set of transitions. Σ^a is assumed to be given by $\Sigma^a = \Sigma^a_{inp} \cup \Sigma^a_{out} \cup \Sigma^a_{\tau}$ of mutually disjoint alphabets.*

We write $(q, j!l, q') \in \delta$ *instead of* $(q, (j, !, l), q') \in \delta$ *and analogously for labels involving* ?. *Moreover we define* $\Sigma^{\mathrm{snd}} = \{l \in \Sigma^c \mid \exists q, q' \in Q \,.\, \exists j \in \mathbb{N} \,.\, (q, j!l, q') \in \delta\}$ *and analogously* Σ^{rcv} *for* $(q, j?l, q') \in \delta$.

In the following we also allow compositions of buffered IOTSs which are not necessarily closed. We use an I/O distinction $\Sigma^a = \Sigma^a_{inp} \cup \Sigma^a_{out} \cup \Sigma^a_\tau$ and treat open I/O-transitions as local actions. By this means, CFSMs with local actions can be used for the translation of finite completely persistent αPIOs. More formally, we consider the class of *IOTSs with asynchronous output labelling* (αIOTS) given by finite I/O-transition systems $A = (L, S, s_0, \Delta)$ with partitioned I/O-labelling L such that $((L, S, s_0, \Delta, \Pi), O^{\triangleright})$ is an αPIO with $\bigcup L = \bigcup O^{\triangleright}$ and $\Delta = \Pi$. The I/O-labelling $L = ((I_1, O_1), \ldots, (I_n, O_n), T)$ consists as usual of input, output and internal labels.

Definition 5.18 (CFSM^e translation) *Let* A *be an* α*IOTS where* $L_A = ((I_1, O_1), \ldots, (I_n, O_n), T)$ *and* $O^{\triangleright}_A = (O_1, \ldots, O_k)$, $k \leq n$. *The* translation $cfsm^e$ from A to a CFSM *is given by* $cfsm^e(A) = (Q, q_0, \Sigma^e, \delta^e)$ *such that* $Q = S_A$ *and* $q_0 = s_{0,A}$. *The alphabet* Σ^e *is defined by* $\Sigma^c = \bigcup\{I_i \mid 1 \leq i \leq k\} \cup \bigcup O^{\triangleright}_A$ *and* $\Sigma^a_{inp} = \bigcup\{I_i \mid k < i \leq n\}$, $\Sigma^a_{out} = \bigcup\{O_i \mid k < i \leq n\}$ *and* $\Sigma^a_\tau = T$. *The transition relation* δ^e *is given by*

$$\forall 1 \leq j \leq k \,.\, (l \in \Sigma^c \wedge (s, j!l, s') \in \delta^e \iff l \in O_j \wedge (s, l, s') \in \Delta_A),$$
$$\forall 1 \leq j \leq k \,.\, (l \in \Sigma^c \wedge (s, j?l, s') \in \delta^e \iff l \in I_j \wedge (s, l, s') \in \Delta_A),$$
$$l \in \Sigma^a \wedge (s, l, s') \in \delta^e \iff l \in \bigcup\{\bigcup((I_i, O_i), T) \mid k < i \leq n\} \wedge (s, l, s') \in \Delta_A.$$

Our translation treats any open transition as a local action, which definitely makes sense for open outputs, due to the unbounded communication buffers in systems of CFSMs. Together with internal transitions theses can be considered as "autonomous" local actions. The understanding of input transitions as local actions, however, leads to an implicit verification assumption. Namely that the open input transitions indeed are used after the system has been composed further. Depending on the concrete verification problem at hand, it might be the case, that the verification result is invalidated by later compositions. Output compatibility is an example of such a property. Considering an input transition as a local action is equivalent to the hiding of the transition on the level of PIOs, IOTSs respectively. But a verification with hidden open transitions is invalidated within later compositions in case the open transitions are not used by the respective communication partner.

Boigelot et al. define a notion of communication protocol that basically corresponds to the syntactic structure of CFSM systems defined above. Protocols are additionally equipped with an alphabet of local actions and a global control transition relation given by the product of the underlying CFSMs without queues. Instead of the direct definition of protocols as given in [**BGWW97**] we give a definition in the sense of CFSM systems, though based on CFSMs with local actions.

Definition 5.19 (CFSM Protocol, cf. [**BGWW97**]) *Let* Sys^e *be a system* $(M_1, \ldots, M_n)$ *of CFSMs* $(Q_i, q_{0,i}, \Sigma^e_i, \delta^e_i)$ *with local actions. A* protocol for Sys^e *is a tuple*

$$P = (Contr, Init, Act, Msg, Que, Trans),$$

where $Contr = Q_1 \times \cdots \times Q_n$, $Init = (q_{0,1}, \ldots, q_{0,n})$, $Act = \bigcup_i \Sigma^a_i$, $Msg = \bigcup_i \Sigma^c_i$, $Que = (Msg|_1)^* \times \cdots \times (Msg|_k)^*$ *where* k *is the number of queues and* $Msg|_i$ *projects* Msg *to the messages stored by queue* i *in* Que *such that the indices used by the local transitions of each machine* M_i *have their appropriate and unique counterpart in* Que. *Finally* $Trans \subseteq Contr \times Op \times Contr$, *where* $Op = Act \cup \mathbb{N} \times \{!, ?\} \times Msg$ *such that* $((c_1, \ldots, c_n), op, (c'_1, \ldots, c'_n)) \in Trans$ *if there exists* $i \in \{1, \ldots, n\}$ *such that* $(c_i, op, c'_i) \in \delta^e_i$ *and* $c_k = c'_k$ *for all* $k \neq i$.

TABLE 5.2. Protocols [**BGWW97**] extend CFSM systems [**CF05**]

CFSM Protocol (Def. 5.19)	CFSM system (Def. 5.7)
$\mathit{Contr} = Q_1 \times \cdots \times Q_n$	$Q_1 \times \cdots \times Q_n$ from Sys
$\mathit{Init} = (q_{0,1}, \ldots, q_{0,n})$	$(q_{0,1}, \ldots, q_{0,n})$ from Sys
$\mathit{Act} = \bigcup_i \Sigma_i^a$	(no local actions)
$\mathit{Msg} = \bigcup_i \Sigma_i^c$	$\Sigma = \bigcup_i \Sigma_i$
$\mathit{Que} = (\mathit{Msg}\vert_1)^* \times \cdots \times (\mathit{Msg}\vert_k)^*$	$(\Sigma^*)^p$ in Γ_{Sys}
$\mathit{Trans} \subseteq \mathit{Contr} \times \mathit{Op} \times \mathit{Contr}$,	$\bigcup_i \delta_i \subseteq \bigcup_i (Q_i \times \mathit{Op} \times Q_i)$,
$\mathit{Op} = \mathit{Act} \cup (\mathbb{N} \times \{!, ?\} \times \mathit{Msg})$	$\mathit{Op} = \mathbb{N} \times \{!, ?\} \times \Sigma$

Table 5.2 shows how the constituents of a protocol are related to the elements of a communicating system. The main difference are local actions and a corresponding extension of the control part of the transition relation for transitions without queue effect. Note that Trans is a global control transition relation. There is not yet any queue effect associated with the gathered local transitions. A protocol is a foremost structural aid to define the syntactic ingredients of a communicating system.

Example 5.20 (Protocol) *In the producer/consumer example Ex. 4.24 we considered PIOs with must transitions only, hence we can consider them as IOTSs; cf. Fig. 4.7. The buffer controller showed an open output transition obviating their translation to a CFSM system. With the definition of extended CFSMs and protocols we may now translate the given systems as depicted in Fig. 5.6 with* $\Sigma^{\mathrm{snd}} = \{\mathsf{put}\}$, $\Sigma^a = \emptyset$ *for* $\mathit{cfsm}^e(\mathsf{Prod})$*; and* $\Sigma^{\mathrm{rcv}} = \{\mathsf{put}\}$, $\Sigma^a_{out} = \{\mathsf{get}\}$ *for* $\mathit{cfsm}^e(\mathsf{BufCtr})$*. From* $\mathit{cfsm}^e(\mathsf{Prod})$ *and* $\mathit{cfsm}^e(\mathsf{BufCtr})$ *we obtain a protocol with* $\mathit{Contr} = \{p_0\} \times \{b_0, b_1\}$, $\mathit{Init} = (p_0, b_0)$, $\mathit{Act} = \{\mathsf{get}\}$, $\mathit{Msg} = \{\mathsf{put}\}$, $\mathit{Que} = \{\mathsf{put}\}^*$ *and a transition relation* $\mathit{Trans} \subseteq \mathit{Contr} \times \mathit{Op} \times \mathit{Contr}$ *as given by Fig. 5.6.*

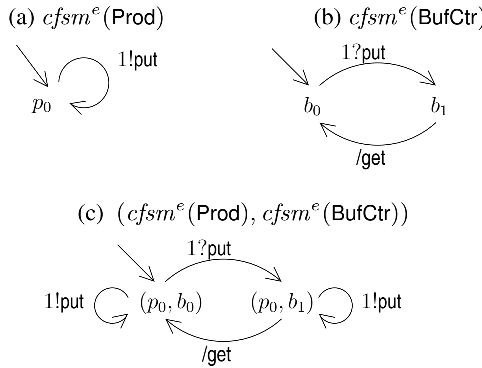


FIGURE 5.6. Extended CFSMs and global control transition relation

The definition of global state space and transition relation for a CFSM protocol proceeds similar to the case of CFSM systems above. The state space of a CFSM protocol P is given by $\Gamma_P = \mathit{Contr} \times \mathit{Que}$. The global transition relation of P is defined by the smallest

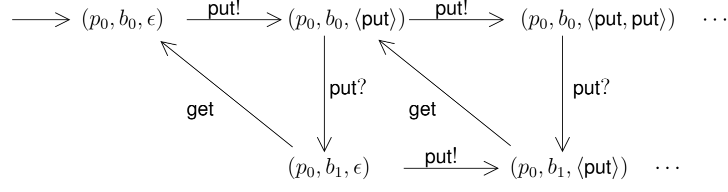


FIGURE 5.7. Global transition relation of CFSM protocol

relation $\rightarrow_P \subseteq \Gamma_P \times (Act \cup (\{!, ?\} \times Msg)) \times \Gamma_P$ such that

(i) $(\vec{c}, j!l, \vec{c}') \in Trans \implies (((\vec{c}, x_1, \dots x_k), l!, (\vec{c}', x'_1, \dots x'_k)) \in \rightarrow_P \wedge$
$(x'_j = x_j l$ and $x'_h = x_h$ for all $h \neq j))$,

(ii) $(\vec{c}, j?l, \vec{c}') \in Trans \implies (((\vec{c}, x_1, \dots x_k), l?, (\vec{c}', x'_1, \dots x'_k)) \in \rightarrow_P \wedge$
$(lx'_j = x_j$ and $x'_h = x_h$ for all $h \neq j))$,

(iii) $(\vec{c}, a, \vec{c}') \in Trans \implies ((\vec{c}, \vec{x}), a, (\vec{c}', \vec{x})) \in \rightarrow_P$.

Example 5.21 (Global transition relation) *The global transition relation for the protocol of* $(cfsm^e(\textit{Prod}), cfsm^e(\textit{BufCtr}))$ *is an infinite-state system as depicted in Fig. 5.7.*

The iots-translation $iots^e(P)$ of a protocol P is defined analogously to the translation $iots(Sys)$ of a communicating system Sys with the difference that the sets of input and output labels are not empty anymore:

1. $L = (I, O, T)$ with $I = \bigcup_i \Sigma^a_{i,inp}$, $O = \bigcup_i \Sigma^a_{i,out}$ and $T = \bigcup_i \Sigma^a_{i,\tau} \cup \bigcup_i \{l^\triangleright \mid l \in \Sigma^{\text{snd}}_i\} \cup \bigcup_i \{l \mid l \in \Sigma^{\text{rcv}}_i\}$,
2. S, s_0 and Δ are obtained by a reordering analogously to the translation $iots(Sys)$.

With $cfsm^e$ and $iots^e$ we are able to translate compositions of buffered IOTSs that are not necessarily closed. The only restriction left concerns rendezvous communication via shared labels. In contrast to the general case we can not allow synchronous communication since communication in CFSM systems is always buffered and hence there is no equivalent for rendezvous communication of buffered IOTSs. However, the restriction seems not to be to severe since a system with both, rendezvous and buffered communication could be transformed into a system with only buffered communication by composing all systems with synchronous communication before analysis. Again we state the correspondence without formal proof. A system $((A_1, O^\triangleright_1), \dots, (A_n, O^\triangleright_n))$ of pairwise composable asynchronous IOTSs is *internally completely asynchronous*, if $(\bigcup L_{A_i} \setminus \bigcup O^\triangleright_i) \cap (\bigcup L_{A_j} \setminus \bigcup O^\triangleright_j) = \emptyset$ for all $i, j \in \{1, \dots, n\}$ with $i \neq j$. Analogous to the correspondence Prop. 5.9 above, we assume no unused labels to simplify the presentation.

Proposition 5.22 (Correspondence) *If a system* $((A_1, O^\triangleright_1), \dots, (A_n, O^\triangleright_n))$ *of pairwise composable asynchronous IOTSs is without unused labels and internally completely asynchronous, then*

$$iots^e(cfsm^e(A_1), \dots, cfsm^e(A_n)) = \bigotimes_i \Omega(A_i, O^\triangleright_i). \qquad \square$$

Example 5.23 (Translation) *Figure 5.8 shows an excerpt of the (infinite-state) IOTSs obtained from the composition* $\Omega(\textit{Prod}, L^\triangleright_{\textit{Prod}}) \otimes \Omega(\textit{BufCtr}, L^\triangleright_{\textit{Ctr}})$ *(cf. Fig. 4.8 in Chap. 4). After reordering the state vectors and relabelling enqueue actions the given transition system indeed coincides with Fig. 5.7.*

For the verification of communication safety, the definition of asynchronous responsiveness Def. 5.14 needs adjustment due to the local actions in extended CFSM systems. The definition of unspecified reception carries over identically, but the definition of ignored reception states and eventually receiving states need to take the local actions in

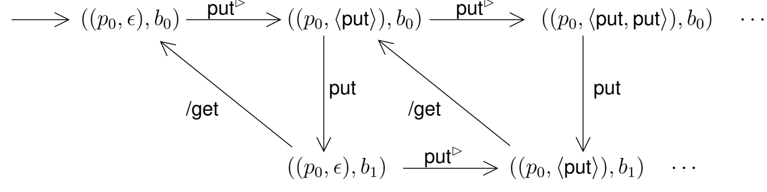


FIGURE 5.8. Composition of buffered IOTSs

$\Sigma^a = \Sigma^a_{inp} \cup \Sigma^a_{out} \cup \Sigma^a_\tau$ into account. The crucial difference lies in the underlying notions of mixed states and snd-reachable states that are extended to cope with local actions labelled by Σ^a as follows.

Let $M = (Q, q_0, \Sigma^e, \delta^e)$ be an extended CFSM with $\Sigma^e = \Sigma^c \cup \Sigma^a$. A state $q \in Q$ is a *mixed state* if it shows both an input as well as an output or a local transition; that is, if there exists $q', q'' \in Q$ such that $(q, l?, q') \in Q$ with $l \in \Sigma^{\mathrm{rcv}}_M$ and $(q, l', q'') \in Q$ with $l' \in \Sigma^{\mathrm{rcv}}_M \cup \Sigma^a$. Now, the extension of Def. 5.10 is straightforward along the definition of CFSM protocols.

Let P be a CFSM protocol for $(M_1, \ldots, M_n)$ extended CFSMs with $k = |Que|$. A state $\gamma = (q_1, \ldots, q_n, x_1, \ldots, x_k) \in \Gamma_P$ is an *ineffective reception configuration*, if there exists $i \in \{1, \ldots, n\}$ such that q_i is a receiving or a mixed state and

$$\forall l \in \Sigma^c \,.\, \forall q'_i \in Q_i \,.\, (q_i, j?l, q'_i) \in \delta^e \implies |x_j| \geq 1 \wedge x_j \neq l\ (Msg|_j)^* \ .$$

If q_i is a receiving state, then γ is an *unspecified reception configuration*, if it is a mixed state we call γ an *ignored reception configuration*.

For the definition of eventually receiving states, all locally reachable states instead of snd-reachable states need to be considered. Note that this includes snd-reachable states. Let $M = (Q, q_0, \Sigma^e, \delta^e)$ be an extended CFSM with $\Sigma^e = \Sigma^c \cup \Sigma^a$ and let $q \in Q$. The *locally reachable states from* q are given inductively by $q \in \mathscr{R}^{\mathrm{loc}}(q)$; and if $q' \in \mathscr{R}^{\mathrm{loc}}(q)$ and there is $q'' \in Q$ such that $(q', l, q'') \in \delta^e$ with $l \in \Sigma^{\mathrm{snd}}_M \cup \Sigma^a$ then $q'' \in \mathscr{R}^{\mathrm{loc}}(q)$. The extension of the definition is again straightforward.

Let $P = (\mathit{Contr}, \mathit{Init}, \mathit{Act}, \mathit{Msg}, \mathit{Que}, \mathit{Trans})$ and $\gamma = (q_1, \ldots, q_n, x_1, \ldots, x_k) \in \Gamma_P$ where $k = |Que|$. The state γ is *eventually receiving*, if for all $j \in \{1, \ldots, k\}$ there exists $i \in \{1, \ldots, n\}$ such that the following holds:

$$x_j \neq \epsilon \implies (\exists (q'_1, \ldots, q'_n, x'_1, \ldots, x'_k) \in \mathscr{R}(\gamma) \,.\, (q'_i \in \mathscr{R}^{\mathrm{loc}}(q_i) \wedge q'_i \text{ is a receiving or a mixed state for } j\ )) \ .$$

With the revised definitions we define asynchronous responsiveness for CFSM protocols and claim the correspondence with ultra-weak comm-safety of the corresponding system of asynchronous IOTSs. A CFSM protocol P is *asynchronously responsive*, if for all $\gamma \in \mathscr{R}(\Gamma_P)$ it holds that γ is eventually receiving and is neither an unspecified nor an ignored reception configuration.

Claim 5.24 *Let $(A_1, \ldots, A_n)$ be a system of pairwise composable internally completely asynchronous IOTSs. $(\Omega(A_1), \ldots, \Omega(A_n))$ is ultra-weakly comm-safe iff the CFSM protocol for $(\mathit{cfsm}^e(A_1), \ldots, \mathit{cfsm}^e(A_n))$ is asynchronously responsive.*

In order to illustrate the basic ideas for the verification of communication safety within compositions of buffered IOTSs we consider a simple example involving a fictitious protocol between a bank server and an ATM.

Example 5.25 (Simple Bank/ATM) *For the IOTSs given in Fig. 5.9 we assume alphabets such that (Bank, Atm) is a system of pairwise composable internally completely asynchronous IOTSs. The ATM sends immediately after the request for pin verification (vp) a request for withdrawal (wd). In case the bank acknowledges the pin to be correct (ok) the*

system proceeds with the permission to give money (gm). Note the simultaneous sending of messages ok and wd.

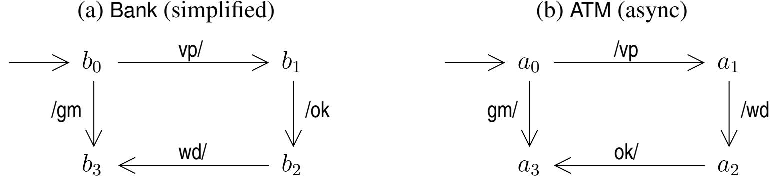


FIGURE 5.9. IOTSs for Bank/ATM protocol with incorrect ATM

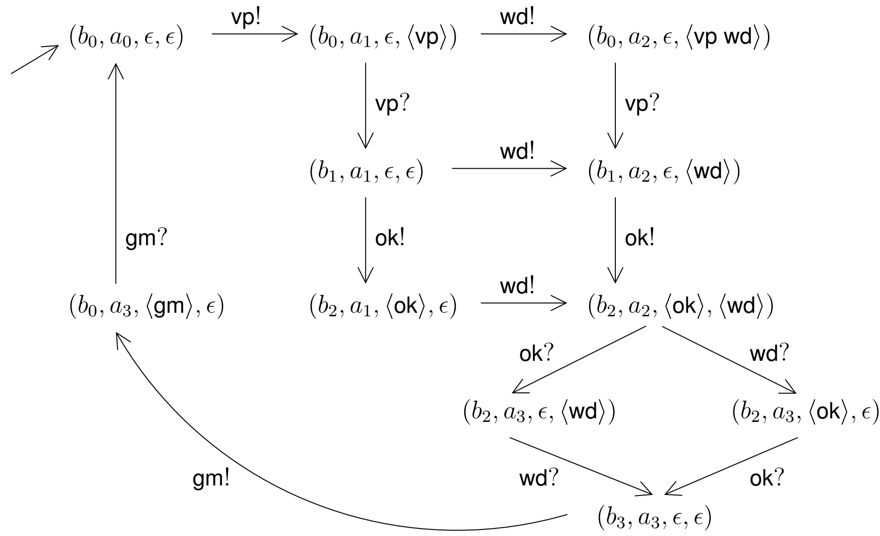


FIGURE 5.10. Global transition relation for $(cfsm^e(\mathsf{Bank}), cfsm^e(\mathsf{Atm}))$

The cfsm^e translation of the given IOTSs results in a CFSM protocol for $(cfsm^e(\mathit{Bank}), cfsm^e(\mathit{ATM}))$ *with global transition relation as depicted in Fig. 5.10. Obviously the state space is finite and we may apply verification techniques for finite-state systems.*

The basic idea for the verification of communication-safety is to verify asynchronous responsiveness for the given CFSM system and conclude ultra-weak comm-safety of the corresponding composition of buffered IOTSs by Prop. 5.16, Claim 5.24 respectively. To show asynchronous responsiveness we need to check that the given system is eventually receiving and free from unspecified and ignored receptions. The system is eventually receiving since for any state with an empty queue there are receiving states reachable solely via loc*-transitions, that is transitions labelled with labels from* $\Sigma^{\mathrm{snd}} \cup \Sigma^a$. *There is exactly one mixed state,* $((b_0, a_1), \epsilon, \mathit{vp})$ *after the first sending of vp!, and this state is free of ignored receptions due to the outgoing receive transition for message vp. Thus the system is free from ignored receptions and it is free from unspecified receptions, since for any receiving state it holds that the possible queue contents can be processed by the outgoing transitions. Therefore the CFSM protocol* $(cfsm^e(\mathit{Bank}), cfsm^e(\mathit{Atm}))$ *is asynchronously responsive and by Prop. 5.16, Claim 5.24 respectively. we conclude that* $\Omega(\mathit{Bank}) \otimes \Omega(\mathit{Atm})$ *is ultra-weakly comm-safe.*

We extend the Bank/ATM example to illustrate a protocol with communication errors due to mixed states within the given local transition systems of Bank and Atm.

Example 5.26 (Bank/ATM mixed) *Consider the Bank/ATM protocol extended by transitions to communicate failed pin verifications as given by Fig. 5.11 and the clip of the corresponding global transition system in Fig. 5.12.*

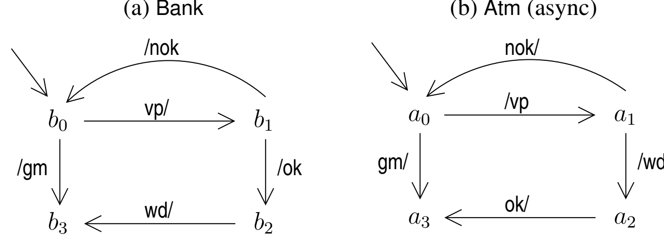


FIGURE 5.11. Extension to communicate failed pin verification

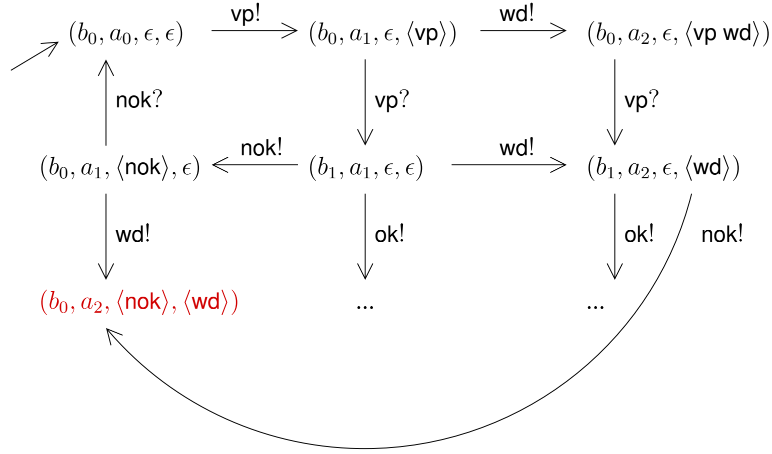


FIGURE 5.12. Clip of global transition system with incorrect ATM

The state $((b_0, a_1), \text{nok}, \epsilon)$ is a mixed state with an outgoing reception of nok *and an outgoing send transition for* wd*. The state is free from ignored receptions, since the queue content* nok *is received along the transition leading to state $((b_0, a_0), \epsilon, \epsilon)$. Though the system is not asynchronously responsive. The state $((b_0, a_2), \text{nok}, \text{wd})$, which was not present in the original global transition system, is a state that is not eventually receiving. The erroneous state is due to the wrong order of sending* wd *and receiving* ok *in the protocol of the* Atm *together with the new output* nok *in the* Bank *protocol. Therefore $\Omega(\text{Bank}) \otimes \Omega(\text{Atm})$ is not ultra-weakly comm-safe.*

The global transition system of the given example is a finite-state system. Hereafter we consider a symbolic verification approach that works for a certain class of infinite-state systems, namely those that are QDD-representable. An extension for the verification of PIOs instead of IOTSs is briefly discussed in Sect. 2.6.

2.4. Queue Content Decision Diagrams (QDD). A Queue-content Decision Diagram (QDD) is a specific finite state machine that can be used for the symbolic representation of queue contents within systems communicating via unbounded FIFO channels. In the following we summarise basic definitions and operations for QDDs from [**BGWW97**] and [**BG99**] and after that illustrate how protocols (i.e. CFSM systems with local actions) extended with QDDs can be used to verify asynchronous compatibility of systems with buffered IOTSs.

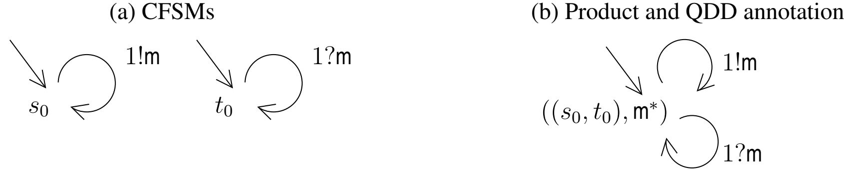


FIGURE 5.13. Extension by QDDs

A *finite-state automata* is a tuple $\mathscr{F} = (\Sigma, S, s_0, \Delta, F)$, where Σ is an alphabet, S is a finite set of states, $s_0 \in S$ an initial state, $\Delta \subseteq S \times \Sigma \cup \{\epsilon\} \times S$ a transition relation and F a set of accepting states. A finite word $w \in \Sigma^*$ with $w = a_1 \ldots a_n$ is accepted by $\mathscr{F}$ if there exists $s_i \in S$ such that $(s_{i-1}, a_i, s_i) \in \Delta$ and $s_n \in F$ for all $1 \leq i \leq n$. The language accepted by $\mathscr{F}$ is given by $\mathcal{L}(\mathscr{F}) = \{w \mid w \text{ is accepted by } \mathscr{F}\}$. The projection of a word w on a subset $\Sigma' \subseteq \Sigma$, written $w|_{\Sigma'}$, is given by removing all symbols in w which are not in Σ'. $\mathscr{F}$ is deterministic if there is no transition labelled ϵ and all outgoing transitions of any reachable state show different labels.
Let $P = (\mathit{Contr}, \mathit{Init}, \mathit{Act}, \mathit{Msg}, \mathit{Que}, \mathit{Trans})$ be a protocol and let $k = |\mathit{Que}|$. A *queue-content decision diagram (QDD)* for P is a deterministic finite-state automata $\mathscr{Q}$ such that for all $w \in \mathcal{L}(\mathscr{Q})$ it holds that $w = w|_{\mathit{Msg}_1} \ldots w|_{\mathit{Msg}_k}$, where Msg_i is the set of messages stored by queue i for $1 \leq i \leq k$.

The queue content within the global state space is represented by QDDs such that each global control state in Contr is extended by a QDD whose accepted language represents the possible queue contents at this global state.

Example 5.27 (QDD) *Figure 5.13 shows a simple system of two CFSMs which communicate via one queue (with identifier* 1*). The product is extended by* m^* *which is a regular expression describing the language accepted by the QDD of this global control state.*

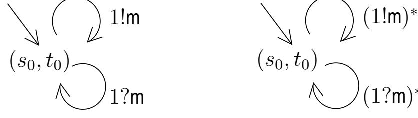
The QDDs are computed along an extended global transition relation. The computation uses specific operations to compute the accepted language $\mathcal{L}(\mathscr{Q})$ for a given global control state $\vec{c} \in \mathit{Contr}$ depending on the queue operation $op \in \{i!m, i?m\}$ obtained from a transition $(\vec{c}', op, \vec{c}) \in \mathit{Trans}$ to $\vec{c}$. Thus, while travelling through the extended control transition system the accepted language assigned to a control state evolves each time the state is visited. The computation terminates if the accepted language of each control state has stabilised (i.e. does not change any more). Stabilisation, and therefore termination is not guaranteed in general. However, if the algorithm terminates, then the final pairs of control states and QDDs are an exact representation of the reachable global state space of the underlying protocol. QDD representable systems satisfy a regularity property. The algorithm is independent from the ordering used for the construction of the accepted languages $\mathcal{L}(\mathscr{Q})$ of the QDDs; cf. [**BGWW97**, Thm. 3].

An important concept used for the computation of QDDs are meta-transitions. For specific patterns in the control transition system, meta-transitions are added which allow to generate a whole set of reachable states in one step. For example a self-loop $(\vec{c}, i!m, \vec{c})$ results in a meta-transition $(\vec{c}, (i!m)^*, \vec{c})$ which is evaluated during QDD construction.

Example 5.28 (Meta-transition) *The example above is extended by meta-transitions as illustrated in Fig. 5.14. The QDD computation for such an extended control transition relation concludes* m^* *as an accepted language for the given control state. Note that an approach without meta-transitions would not terminate for the given example since the accepted language evolves infinitely. Moreover, note that the accepted language would not differ, if the receiver in Fig. 5.13 would wait for an input different from* m*.*

Therefore, meta-transitions are an important concept to enable the construction of finite representations of an infinite state space. Boigelot et al. [**BGWW97**] propose three

(a) Control transition relation (left) and meta-transitions (right)



(b) Global transition relation without QDDs

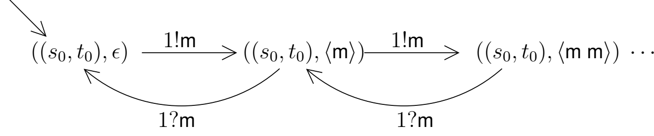


FIGURE 5.14. Meta-transitions and infinite state space

kinds of meta-transitions: repeatedly receiving messages (1), repeatedly sending messages (2) and repeatedly receiving messages on one queue followed by sending messages on a different queue (3). Assuming that reachability skips over local actions (i.e. transitions without queue effect), meta-transitions are added to the global control transition systems as follows:

(1) $(\vec{c}, (i?w)^*, \vec{c}')$ with $w = m_1 \dots m_n$,
if $\vec{c}'$ is reachable from $\vec{c}$ by transitions labelled $i?m_h$ for $1 \leq h \leq n$,
(2) $(\vec{c}, (i!w)^*, \vec{c}')$ with $w = m_1 \dots m_n$,
if $\vec{c}'$ is reachable from $\vec{c}$ by transitions labelled $i!m_h$ for $1 \leq h \leq n$,
(3) $(\vec{c}, (i?w_1; j!w_2)^*, \vec{c}')$ for $i \neq j$ with $w_1 = m_1 \dots m_n$ and $w_2 = m_1 \dots m_r$,
if $\vec{c}'$ is reachable from $\vec{c}$ by transitions labelled $i?m_h$ for $1 \leq h \leq n$ followed by transitions labelled $j!m_g$ for $1 \leq g \leq r$.

The language definitions explained hereafter precisely specify, on the one hand the effect of ordinary transitions, and on the other hand, the effect of meta-transitions on the computation of the QDDs. In total there are five cases which need to be distinguished. The corresponding QDD operations computing the effect of global control transitions on the evolution of the associated QDDs are given in [**BG99**] and implemented in a C library [**Boi**] that could be included in a model checker for CFSM systems.

Let $P = (\mathit{Contr}, \mathit{Init}, \mathit{Act}, \mathit{Msg}, \mathit{Que}, \mathit{Trans})$ be a protocol. Let $\vec{c} \in \mathit{Contr}$ and $\mathscr{Q}$ the QDD associated with $\vec{c}$. Let Trans^μ be a relation obtained from Trans by the addition of meta-transitions. Let $(\vec{c}, op, \vec{c}') \in \mathit{Trans}^\mu$ where $op \in \{i?m, i!m\} \cup \{(i?w)^*, (i!w)^*, (i?w_1; j!w_2)^*\}$. Then the language $\mathcal{L}_{op}(\mathscr{Q})$ accepted by the QDD for $\vec{c}'$ with respect to the label op of incoming (meta-) transitions is given by:

(1) $\mathcal{L}_{i!m}(\mathscr{Q}) = \{w'' \mid \exists w' \in \mathcal{L}(\mathscr{Q}) \,.\, w''|_{M_i} = w'|_{M_i} m \wedge \forall j \neq i \,.\, w''|_{M_j} = w'|_{M_j}\}$,
$\mathcal{L}_{i?m}(\mathscr{Q}) = \{w'' \mid \exists w' \in \mathcal{L}(\mathscr{Q}) \,.\, w'|_{M_i} = m w''|_{M_i} \wedge \forall j \neq i \,.\, w''|_{M_j} = w'|_{M_j}\}$,

(2) $\mathcal{L}_{(i!w)^*}(\mathscr{Q}) = \{w'' \mid \exists w' \in \mathcal{L}(\mathscr{Q}), k \geq 0 \,.$
$w''|_{M_i} = w'|_{M_i} w^k \ \wedge\ \forall j \neq i \,.\, w''|_{M_j} = w'|_{M_j}\}$,
$\mathcal{L}_{(i?w)^*}(\mathscr{Q}) = \{w'' \mid \exists w' \in \mathcal{L}(\mathscr{Q}), k \geq 0 \,.$
$w'|_{M_i} = w^k w''|_{M_i} \ \wedge\ \forall j \neq i \,.\, w''|_{M_j} = w'|_{M_j}\}$,

(3) $\mathcal{L}_{(i?w_1;j!w_2)^*}(\mathscr{Q}) = \{w'' \mid \exists w' \in \mathcal{L}(\mathscr{Q}), k \geq 0 \,.$
$w'|_{M_i} = w_1^k w''|_{M_i} \wedge w''|_{M_j} = w'|_{M_j} w_2^k \wedge \forall h \notin \{i,j\} \,.\, w''|_{M_h} = w'|_{M_h}\}$.

Definition (1) treats transitions with enqueue or dequeue actions, while (2) and (3) explain the effect of meta-transitions. The conditions are taken from [**BG99**, pp. 242, 244].

2.5. QDDs and Asynchronous Communication Safety. If a given CFSM protocol is QDD-representable, then there exists a finite representation of its global transition relation which associates a QDD with each control state. The QDD represents the contents of the queues within this state. Asynchronous compatibility of systems of buffered IOTSs requires, given a state with some non-empty queues, a succeeding sequence of receive transitions that matches the given queue contents. If this content is represented by a QDD we have, instead of a sequence of messages stored within the particular queue, regular expressions such as $(\mathsf{m}^* \ \mathsf{x} \ \mathsf{n}^*)$, meaning that there are arbitrary many messages m stored topmost of this queue, and after that a message x, etc. A compatible communication partner is then required to receive arbitrary many messages m (and after that x, etc.), thus we seek transitions that match star expressions like m^* which is exactly provided by meta transitions for repeated inputs.

For the concrete verification we may apply the approach described above, using the notion of asynchronous responsiveness of CFSM protocols. Again we need small extensions of the underlying definitions taking the meta-transitions for inputs into account. We assume that the considered CFSMs are annotated with meta-transitions already (cf. Fig. 5.15 for an example). Instead of queue contents, we need to consider the projections of the corresponding QDD. We use the notation qdd_j to refer to the projection of a QDD qdd, describing the jth queue of a global state $\gamma \in \Gamma_P$ of a CFSM protocol P; with $nxt(qdd_j)$ we refer to the topmost element of qdd_j, that is either a single message m or an expression m^*.

Let P be a CFSM protocol for $(M_1, \ldots, M_n)$ extended CFSMs with $k = |Que|$. A state $\gamma = (q_1, \ldots, q_n, x_1, \ldots, x_k) \in \Gamma_P$ is an *ineffective reception configuration*, if there exists $i \in \{1, \ldots, n\}$ such that q_i is a receiving or a mixed state and

$$\forall l \in \Sigma^c \,.\, \forall q_i' \in Q_i \,.\, (q_i, j?l, q_i') \in \delta^e \implies qdd_j \neq \epsilon \wedge nxt(qdd_j) \neq l \,, \tag{1}$$

$$\forall l \in \Sigma^c \,.\, \forall q_i' \in Q_i \,.\, (q_i, (j?l)^*, q_i') \in \delta^e \implies qdd_j \neq \epsilon \wedge (nxt(qdd_j) \neq l \vee nxt(qdd_j) \neq l^*) \,. \tag{2}$$

A state $q \in Q$ of an extended CFSM $M = (Q, q_0, \Sigma^e, \delta^e)$ with meta-transitions is a *receiving or mixed state for* $j \in \mathbb{N}$ if there exists $l \in \Sigma_M^{\mathrm{rcv}}$, $q' \in Q$ such that $(q, j?l, q') \in \delta^e$ or $(q, (j?l)^*, q') \in \delta^e$. Let P be a CFSM protocol for $(M_1, \ldots, M_n)$ extended CFSMs with $k = |Que|$. Let $\gamma = (q_1, \ldots, q_n, qdd_1, \ldots, qdd_k) \in \Gamma_P$. The state γ is *eventually receiving*, if for all $j \in \{1, \ldots, k\}$ there exists $i \in \{1, \ldots, n\}$ such that the following holds:

$$qdd_j \neq \epsilon \implies (\exists (q_1', \ldots, q_n', qdd_1', \ldots, qdd_k') \in \mathscr{R}(\gamma) \,.\, (q_i' \in \mathscr{R}^{\mathrm{loc}}(q_i) \wedge q_i' \text{ is a receiving or a mixed state for } j \,)) \,.$$

By this means, the notion of asynchronous responsiveness is completely adjusted to a setting with meta-transitions and its verification may proceed as before. In order to illustrate these basic ideas for QDD-based verification of communication-safety, we first reconsider the Bank/ATM example Ex. 5.25 from above.

Example 5.29 (Simple Bank/ATM with QDD) *The cfsm translation of the given IOTSs results in a CFSM system* $(cfsm^e(\mathsf{Bank}), cfsm^e(\mathsf{ATM}))$ *with global transition relation as depicted in Fig. 5.10. Obviously the state space is finite and we could equally well apply verification techniques for finite-state systems. However, the system can also be used as a simple example for the verification of communication safety with a QDD-based representation of queue-contents. Since there are no loops or cycles which satisfy the conditions for meta-transitions we may understand the queue contents of the given transition system directly as words accepted by the QDD associated to the corresponding control states. For example, the projection of the QDD for state* $(b_0, a_2, \epsilon, \langle \mathsf{vp\ wd} \rangle)$ *to the second queue accepts the words described by the regular expression* $(\mathsf{vp\ wd})$.

To show asynchronous responsiveness we need to check that the given system is eventually receiving and free from unspecified and ignored receptions. A non-trivial QDD corresponds to a potentially non-empty queue. By this, the argumentation proceeds analogously to Ex. 5.25. The only difference is the QDD based check of the condition $qdd_j \neq \epsilon \wedge nxt(qdd_j) \neq l$ *for unspecified and ignored receptions respectively.*

As a simple but illustrative infinite-state example we consider the IOTSs of a producer and a buffer controller as introduced before. Remember that the system is not only infinite-state but also open, thus it is mandatory to use the extended definitions of CFSM systems with local actions and the extended IOTS translations $cfsm^e$ and $iots^e$.

Example 5.30 (Infinite-state) *The global transition system for the communicating system* $(cfsm^e(\textsf{Prod}), cfsm^e(\textsf{BufCtr}))$ *as depicted in Fig. 5.6 is an infinite-state system with open output transitions. Due to the unbounded output queue of the producer the system may infinitely send items without ever receiving one. Exactly these kind of cycles (loops) are candidates for the introduction of meta-transitions. Figure. 5.15 shows the addition of two meta-transitions. The meta-transition for* Prod *is due to the self loop in the original system and the meta-transition for* BufCtr *is due to the repeated reception of* put *interleaved with the local action* get*. The symbolic representation of the corresponding global transition relation is depicted in Fig. 5.16. The meta-transitions for sending the message* put *allow to derive a QDD which accepts the language described by* put* *as a symbolic representation of the queue content for the control states* (p_0, b_0) *and* (p_0, b_1)*.*

FIGURE 5.15. Extended CFSMs with meta-transitions

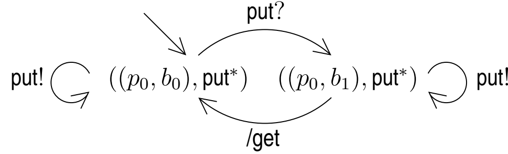


FIGURE 5.16. Finite representation of an infinite state space

The verification requires to check for unspecified or ignored receptions and eventually receiving states. The last requirement indeed holds, since for the QDDs in both states the initial state is locally reachable and the state b_0 *is an appropriate mixed state (due to the reception of* put*). For the remaining requirements we have only a mixed state (again in* b_0*) and no receive state. The CFSM* $cfsm^e(\textsf{BufCtr})$ *shows a meta-transition labelled* $(\textsf{put?})^*$*, thus in case of a non-trivial QDD in the corresponding global state we need to check the second requirement of ineffective receptions above. The state* $((p_0, b_0), \textsf{put}^*)$ *yields a non-trivial QDD with* $nxt(qdd) = \textsf{put}^*$ *and hence the input transitions of the state* b_0 *in* $cfsm^e(\textsf{BufCtr})$ *must provide a corresponding meta-transition. Since such a transition exists the given system is asynchronously responsive.*

2.6. On Extensions for the Verification of PIOs. PIOs extend I/O-transition systems with a set of persistent transitions. Formally, the persistent transitions of a PIO are a subset of the transition relation of its underlying IOTS and thus a kind of additional information that can be incorporated into the CFSM view of systems with FIFO buffered communication. In the following, we exemplify this claim for the case of CFSM systems without local actions.

Given a CFSM system (Q, q_0, Σ, δ), we denote the subset of *persistent CFSM transitions* by $\pi(\delta)$. The subset is either assumed to be specified or given by a translation from PIOs. In the former case we extend the definition of CFSMs to obtain a correspondence between systems of buffered PIOs and thereby extended CFSM systems. In the latter case we define $\pi(\delta)$ as part of the CFSM translation of PIOs. In both cases, however, the additional information is available in the form of a transition subset $\pi(\delta)$.

The support for the verification of asynchronous comm-safety for PIOs using a definition of asynchronous responsiveness for CFSM systems can be extended to take persistent transitions into account in a straightforward way. The definition of ineffective receptions uses $\pi(\delta)$ instead of δ, i.e., under the assumptions of Def. 5.10,

$$\forall l \in \Sigma_i \,.\, \forall q_i' \in Q_i \,.\, (q_i, j?l, q_i') \in \pi(\delta_i) \implies (|x_j| \geq 1 \wedge x_j \neq l\, \Sigma^*) ,$$

with the intuitive meaning that there should be a persistent reception, instead of an arbitrary reception, that can be used to dequeue the next element. The extension of eventually receiving states is based on an extension of the snd-reachable states. The reachability path must use persistent transitions only. If we denote by $\mathscr{R}^{\mathrm{snd}}_{\pi(\delta)}(q)$ the states, reachable using persistent snd-transitions in $\pi(\delta)$, then the condition for eventually receiving states is extended, with assumptions as given by Def. 5.13 merely by replacing $\mathscr{R}^{\mathrm{snd}}(q_i)$:

$$\begin{aligned} x_j \neq \epsilon \implies (\exists (q_1', \ldots, q_n', x_1', \ldots, x_p') \in \mathscr{R}(\gamma) \,.\, (q_i' \in \mathscr{R}^{\mathrm{snd}}_{\pi(\delta)}(q_i) \wedge \\ q_i' \text{ is a receiving or a mixed state for } j)) \,. \end{aligned}$$

In a next step, the correspondence with asynchronous comm-safety of buffered PIO systems in Prop. 5.16 must be established using the new definitions. We believe that a succeeding extension for CFSM systems with local actions and for the symbolic verification with QDDs works analogously to the stepwise extensions as carried out in the course of this chapter for the case of I/O-transition systems.

CHAPTER 6

Frames for the Specification of Component Behaviours

1. Frame Specifications for Port-Based Components	**99**
1.1. Correct Components	100
1.2. Frame Analysis	101
2. Compressing Proxy Revisited	**103**
2.1. Frame Specification and Correctness	103
2.2. Communication-Safety	105
3. Supporting Component-Based Development	**107**
3.1. Top-Down and Bottom-Up Design	107
3.2. Evolution in Hierarchical Systems	108
3.3. Weak- and Ultra-Weak Communication-Safety	108
4. Port-Based Frame Verification of Communication-Safety	**110**
5. Related Work	**112**

In Chap. 2 we defined a formal model for port-based components with behaviours. Behaviours are formalised by I/O-transition systems and represent the implementation of a component. Within this chapter we extend the given model by a specification layer using the theory developed in Chap. 4. We illustrate our extension using the compressing proxy system that was already used to introduce component behaviours in Chap. 2. In this context we discuss implementation correctness, refinement and communication-safety. After that, we show that our model supports important characteristics of component-based development such as top-down and bottom-up design approaches. Finally, we discuss a port-based approach for the verification of compatibility and communication-safety based on frames and port protocols.

1. Frame Specifications for Port-Based Components

In correspondence with the terminology used by SOFA [**BHP06**], we call the specification of a component's behaviour a "frame" and integrate these kinds of specifications within our metamodel as depicted in Fig. 6.1. Any component is assumed to be equipped with a frame. There is an association between behaviours and frames which represents a correctness relation to be formally defined below. The behaviour of a correct component is associated with the frame of the particular component. Since components are not necessarily correct, however, their behaviour might not be in the correctness relation and hence does not refer to any frame. In contrast, since frames represent a behavioural specification, there is usually a set of behaviours which could be considered a valid implementation of some given frame.

Frame specification use only external I/O-labels and the anonymous τ action for the abstract modelling of internal choice.

Definition 6.1 (Frame) *The* frame *of a component* $C \in \mathrm{Cmp}$ *is an input-persistent I/O-transition system* $frm(C) = ((I, O, T), S, s_0, \Delta, \Pi)$ *with* $I = \bigcup\{p.msg(prv(P)) \mid p : P \in ports(C)\}$, $O = \bigcup\{p.msg(req(P)) \mid p : P \in ports(C)\}$ *and* $T = \emptyset$*.*

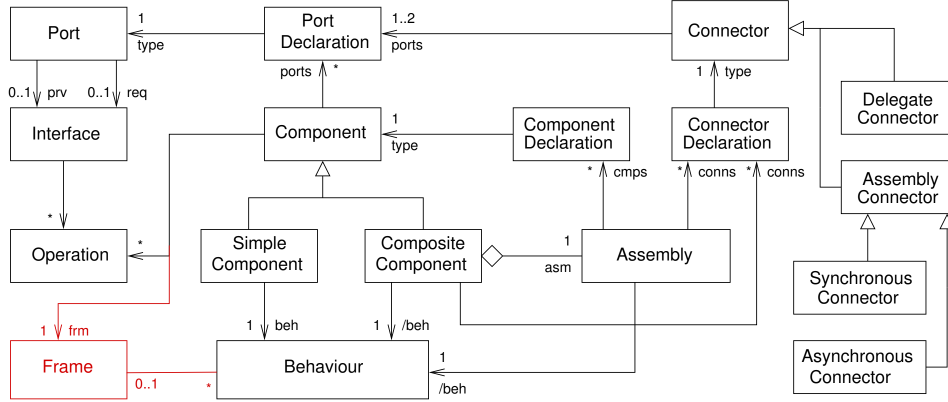


FIGURE 6.1. Extension of the metamodel (cf. Fig. 2.1)

Note that the port prefix of the I/O-labels of a frame allows to create an appropriate partitioned I/O-labelling if needed for the formal treatment of buffered communication. As long as this is not the case we continue to use unpartitioned I/O-labellings for better readability.

1.1. Correct Components. Input-persistent I/O-transition systems (PIOs) specify, besides the transitional behaviour of a component, which of its transitions are persistent. Behaviours, in contrast, show only the transitional behaviour of a component. In the more general formalism of modal automata [**LT88**] an implementation is an automata with must-transitions only. In the context of PIOs this corresponds to complete persistence, i.e. to $\Delta = \Pi$. Hence implementations are perfectly described by I/O-transition systems (L, S, s_0, Π). Put the other way around behaviours can be readily applied to a definition of correct implementations along the refinement relation for PIOs if they are translated to a PIO with $\Delta = \Pi$ before. The *pio-translation of an I/O-transition system* is defined by $pio(L, S, s_0, \Delta) = (L, S, s_0, \Delta, \Delta)$.

Definition 6.2 (Correct implementation) *Let* $C \in \mathrm{Cmp}$. *The class of* correct implementations *of* C *is given by* $[\![frm(C)]\!] = \{(L, S, s_0, \Delta) \mid pio(L, S, s_0, \Delta) \sqsubseteq^{\mathrm{bb}} frm(C)\}$. *Component* C *is correct (correctly implemented) if* $beh(C) \in [\![frm(C)]\!]$.

Showing that a component is correct is fundamental to the verification of component-based designs no matter if a top-down or a bottom-up approach was chosen for the design of the given system. For bottom-up approaches it allows to conclude the correctness of the composed implementations w.r.t. some global frame and for top-up approaches it allows to conclude that the composed frames are a refinement of such a frame. Section 3 further elaborates on this topic. In the following we first need to be more precise on mechanisms for the composition of frames.

Frame specifications make use of the same static structures as component behaviours. Therefore we apply the same assembly mechanism to define the composition of frames. Analogously to behaviours we need to extend the notion of frames to component declarations and clarify what the frame of a connector declaration is. Then, we may use the match relabellings α and σ as defined in Chap. 2 to ensure proper synchronisation between the composed entities. The behaviour of a synchronous connector is an empty transition system which acts as a neutral element for the composition of transition systems and the behaviour of an asynchronous connector is defined by the composition of two queues associated with the ports of the given connector. Since both kinds of connectors do not allow for user-defined behaviours, for example a message ordering different from FIFO, there is no reason to consider user-defined frames of connectors. Thus, the *frame of a connector declaration* $k : K \in \mathrm{ConnDcl}$ is given by translation of its buffering behaviour to an

input-persistent I/O-transition system, that is $\mathit{frm}(k : K) = \mathit{pio}(\mathit{buf}(k : K))$. Note that $\mathit{buf}(k : K)$ does not show internal labels due to the definition of queue IOTSs. Therefore, the frame of a connector (declaration) is indeed a frame in the sense that it does not show any internal but only input and output labels. The frame of a component declaration $c : C \in \mathrm{CmpDcl}$ is given by $\mathit{frm}(c : C) = c\,.\,\mathit{frm}(C)$. The following lemma states the equivalence of the given translation and a direct definition using the PIO encoding of FIFO queues (cf. Def. 4.21) for later reference. The claim follows directly from the definitions.

Lemma 6.3 *Let $k : K$ be an asynchronous connector with $\mathit{ports}(K) = \{p : P, q : Q\}$. Then $\mathit{pio}(\mathit{buf}(k : K)) = k\,.\,(Q^{\triangleright}_{msg(req(P))} \otimes Q^{\triangleright}_{msg(req(Q))})$, where $Q^{\triangleright}$ is the PIO encoding of FIFO queues w.r.t. required messages of P and Q resp.* □

Definition 6.4 (Frame assembly) *The* frame *of an assembly a is given by*

$$\mathit{frm}(a) = (\textstyle\bigotimes_{c:C \in cmps(a)} (\mathit{frm}(c : C)\sigma\alpha) \otimes \bigotimes_{k:K \in conns(a)} \mathit{frm}(k : K))\ .$$

Composite components are assumed to be equipped with a frame just like simple components. Correctness as required by Def. 6.2 above then depends on the relation between the assembly behaviour of the component and its frame which is consistent with the definition of composite components in Chap. 2. With the following table we provide a brief conceptual overview in order to summarise and relate our behavioural definitions with the frame definitions for components, connectors and assemblies.

Behaviour	Frame
$\mathit{beh}(C) = $ IOTS	$\mathit{frm}(C) = $ PIO
$\mathit{buf}(K) = \mathit{que}(P) \otimes \mathit{que}(Q)$	$\mathit{frm}(K) = \mathit{pio}(\mathit{buf}(K))$
$\mathit{beh}(a) = \bigotimes \mathit{beh}(C)\xi \otimes \bigotimes \mathit{buf}(K)$	$\mathit{frm}(a) = \bigotimes \mathit{frm}(C) \otimes \bigotimes \mathit{frm}(K)$
$\mathit{beh}(CC) = \mathit{beh}(a)\rho$	$\mathit{frm}(CC) = $ PIO

The only difference is in the definitions for composite components. The behaviour of assemblies and composite components is derived from the behaviours of the subcomponents. The frame, in contrast, is derived only for assemblies. For composite components we still expect an explicitely given frame specification; cf. Def. 6.2. Note that $\mathit{frm}(a)$ usually shows internal transitions as a result from the synchronisation of component and connector frames. This is in contrast to the internal transitions of the frame of a composite component which is, analogously to simple components, supposed to show anonymous τ actions only.

1.2. Frame Analysis. Having a frame composition mechanism at hand, the compatibility properties studied on the level of PIOs can directly be transferred to the level of frames. Then, by the notion of correct components defined above, an analysis of compatibility, communication-safety (comm-safety) respectively on the level of frame specifications allows to conclude comm-safety of component behaviours. Remember that comm-safety is the n-ary version of output compatibility and comes in three flavours, strong, weak and ultra-weak comm-safety corresponding to the respective kind of output compatibility. Since blackbox refinement preserves weak and ultra-weak output compatibility we have the following.

Proposition 6.5 (Frame analysis) *Let $a \in \mathrm{Asm}$ such that for all $c : C \in \mathit{cmps}(a)$, $\mathit{beh}(c : C) \in [\![\mathit{frm}(c : C)]\!]$. If $\mathit{frm}(a)$ is weakly (ultra-weakly) comm-safe, then $\mathit{beh}(a)$ is weakly (ultra-weakly) comm-safe.*

PROOF. By definition $\mathit{frm}(a)$ is a composition of PIOs. Thus by definition of correctness, $\mathit{pio}(\mathit{beh}(c : C)) \sqsubseteq^{\mathrm{bb}} \mathit{frm}(c : C)$ for all $c : C \in \mathit{cmps}(a)$. The connector transition systems for frame and behaviour assemblies are equivalent due to the direct translation from connector behaviour to connector frame (cf. Lem. 6.3). Hence the assumptions of Prop. 4.35 hold and the claim follows by n applications of Prop. 4.35. □

Strong compatibility is not preserved by blackbox-refinement and hence strong comm-safety does not carry over. In order to get an analogous result, a strong variant of the blackbox refinement that does not allow insertion and removal of internal transitions must be used.[1]

Frame analysis using Prop. 6.5 allows to infer properties of an implementation from an analysis of its specification, i.e. from an analysis on the frame level of the component-based system. In general, results from frame analysis can be expected to carry over for any property which is preserved by blackbox refinement between PIOs. For instance, Chap. 4 examined the preservation of safety properties using a greybox variant of blackbox refinement. Greybox refinement is related to our notion of correct implementations by Thm. 4.47 and Thm. 4.48. We consider assemblies with hidden enqueue actions.

Proposition 6.6 (Safety properties) *Let* $a \in \mathrm{Asm}$ *such that for all* $c : C \in \mathit{cmps}(a)$*,* $\mathit{beh}(c : C) \in [\![\mathit{frm}(c : C)]\!]$*. Let* $H^{\rhd} = \bigcup\{l^{\rhd} \mid l^{\rhd} \in T_{\mathit{frm}(a)}\}$ *and let* P_{sf} *be a safety property for* $\mathit{frm}(a) \setminus H^{\rhd}$*. If* $\mathit{frm}(a) \setminus H^{\rhd} \models P_{\mathrm{sf}}$ *then* $\mathit{beh}(a) \setminus H^{\rhd} \models P_{\mathrm{sf}}$*.*

In order to simplify the proof's presentation, we will use directly behaviours and frames of component types, skip a detailed discussion with relabellings σ and α for the proper synchronisation within assemblies (cf. Def. 2.22), and consider completely buffered PIOs. The latter corresponds to components that communicate by asynchronous connectors only. Let $\mathit{sync}(a) \subseteq \mathit{cmps}(a)$ and $\mathit{async}(a) \subseteq \mathit{cmps}(a)$ be the subsets of synchronously and asynchronously communicating components in assembly a. First we proof the following lemma.

Lemma 6.7 *Let* $a \in \mathrm{Asm}$ *such that for all* $c : C \in \mathit{cmps}(a)$*,* $\mathit{beh}(c : C) \in [\![\mathit{frm}(c : C)]\!]$*. Let* $H^{\rhd} = \bigcup\{l^{\rhd} \mid l^{\rhd} \in T_{\mathit{frm}(a)}\}$*. Then* $\mathit{beh}(a) \setminus H^{\rhd} \sqsubseteq^{\mathrm{gb}} \mathit{frm}(a) \setminus H^{\rhd}$*.*

PROOF OF LEMMA. By definition of assembly frames, associativity and commutativity of the product operator, and definition of buffered PIOs:

$$\begin{aligned}
&\mathit{frm}(a) \setminus H^{\rhd} = (\textstyle\bigotimes_{c:C \in \mathit{cmps}(a)} \mathit{frm}(C) \otimes \bigotimes_{k:K \in \mathit{conns}(a)} \mathit{frm}(K)) \setminus H^{\rhd} \\
&= (\textstyle\bigotimes_{c:C \in \mathit{sync}(a)} \mathit{frm}(C) \otimes \bigotimes_{c:C \in \mathit{async}(a)} \Omega(\mathit{frm}(C))) \setminus H^{\rhd} \\
&= \textstyle\bigotimes_{c:C \in \mathit{sync}(a)} \mathit{frm}(C) \otimes \bigotimes_{c:C \in \mathit{async}(a)} (\Omega(\mathit{frm}(C)) \setminus H^{\rhd}) \\
&= \textstyle\bigotimes_{c:C \in \mathit{sync}(a)} \mathit{frm}(C) \otimes \bigotimes_{c:C \in \mathit{async}(a)} (\Omega(\mathit{frm}(C))\xi).
\end{aligned}$$

For the third step, note that $H^{\rhd}$ comprises only enqueue labels and thus the hiding does not affect synchronously communicating components. As frames do not show internal labels we may replace the hiding of enqueue labels by a hiding of all internal labels using ξ in the fourth step. By the correctness assumption $\mathit{beh}(c : C) \in [\![\mathit{frm}(c : C)]\!]$, the invariance of blackbox refinement w.r.t. ξ (cf. Lem. 4.5), and the transfer property of blackbox refinement (cf. Cor. 4.29) it holds that

$$\textstyle\bigotimes_{c:C \in \mathit{sync}(a)} \mathit{beh}(C)\xi \sqsubseteq^{\mathrm{bb}} \bigotimes_{c:C \in \mathit{sync}(a)} \mathit{frm}(C) \text{ and}$$
$$\textstyle\bigotimes_{c:C \in \mathit{async}(a)} \Omega(\mathit{beh}(C))\xi \sqsubseteq^{\mathrm{bb}} \bigotimes_{c:C \in \mathit{async}(a)} \Omega(\mathit{frm}(C))\xi.$$

Now, theorems 4.47 and 4.48 are applicable and we obtain

$$\textstyle\bigotimes_{c:C \in \mathit{sync}(a)} \mathit{beh}(C)\xi \sqsubseteq^{\mathrm{gb}} \bigotimes_{c:C \in \mathit{sync}(a)} \mathit{frm}(C) \text{ and}$$
$$\textstyle\bigotimes_{c:C \in \mathit{async}(a)} \Omega(\mathit{beh}(C))\xi \sqsubseteq^{\mathrm{gb}} \bigotimes_{c:C \in \mathit{async}(a)} \Omega(\mathit{frm}(C))\xi.$$

Thus, by compositionality of greybox refinement (cf. Cor. 4.46), it follows that

$$\textstyle\bigotimes_{c:C \in \mathit{sync}(a)} \mathit{beh}(C)\xi \otimes \bigotimes_{c:C \in \mathit{async}(a)} \Omega(\mathit{beh}(C))\xi \sqsubseteq^{\mathrm{gb}} \mathit{frm}(a) \setminus H^{\rhd}.$$

As the left-hand side is obtained from $\mathit{beh}(a) \setminus H^{\rhd}$ analogously to the case of $\mathit{frm}(a) \setminus H^{\rhd}$ above, the claim follows. □

[1] For example the PIO-correspondence for strong modal refinement of MIOs [**BMSH10**].

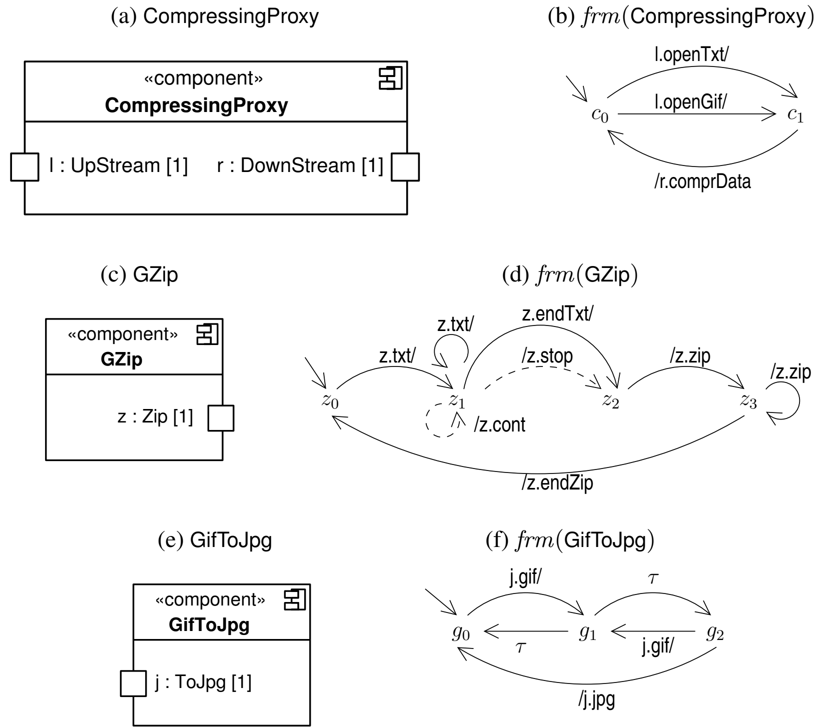


FIGURE 6.2. Frame specifications

PROOF OF PROPOSITION. By Lem. 6.7, it holds that $beh(a) \setminus H^{\triangleright} \sqsubseteq^{\mathrm{gb}} frm(a) \setminus H^{\triangleright}$. Hence, the claim follows by Thm. 4.53 if $beh(a)$ is without terminal states. The general case holds by a generalisation of Thm. 4.53; cf. p. 71. □

2. Compressing Proxy Revisited

Within this section we aim at illustrating the concepts of our component model extended with frame specifications. First, we will introduce frame specifications for the components constituting the compressing proxy system known from Chap. 2 and after that discuss component correctness for simple and composite components. Finally we will discuss analysis and preservation of comm-safety within the given system.

2.1. Frame Specification and Correctness. We start with the compressing proxy component introduced in Chap. 2 (Sect. 1.2) and consider its originally required functionality. The component should receive uncompressed textual and graphical data upstream and send compressed data further downstream. An immediate design to meet these requirements is shown in Fig. 6.2a/6.2b. For the port interfaces we assume the port UpStream to be equipped with a provided interface $\mathsf{USP} = \{\mathsf{openTxt}, \mathsf{openGif}\}$ and the port DownStream with a required interface $\mathsf{DSR} = \{\mathsf{comprData}\}$.

Next, assume we have components GZip and GifToJpg available as tools for textual and graphical compression as depicted in Fig. 6.2c/6.2d and Fig. 6.2e/6.2f. In order to employ these tools for the implementation of ComprProxy we will need to design a component which links their behaviour with the required functionality of the compressing proxy component. Since component-based design aims at independence from implementation details this addresses the frames of the given components which are assumed to be given by the transition systems (PIOs) in Fig. 6.2d and Fig. 6.2f. The interfaces for the ports of these

components are assumed to be given according to the labels used in the respective frame specification.

The frame specification of the zip compression component specifies a compression tool which retrieves streamed textual data until z.endTxt is triggered. In this case a corresponding zip archive is sent back, again using a stream with a corresponding endZip message. An optional feature is the possibility to interrupt the reception of textual data by sending z.stop, for instance to avoid an internal buffer overflow of the zip component. The feature is optional due to the specification of z.stop as being not persistent (indicated by a dashed transition arrow). Support of clients which require explicit acknowledgements to continue with text transmission is modelled by the transition labelled z.cont. Acknowledgements again are an optional feature.

In contrast to the frame of GZip, the frame specification for the component GifToJpg is completely persistent. Though there is a difference between behaviours and frame. A frame is not allowed to show transitions with internal labels. Instead frames always hide these kind of labels using the anonymous internal label τ which makes different internal transitions indistinguishable. By this means the size of frame specification may often be reduced significantly along the minimisation procedure of observational equivalence, sometimes even up to the complete removal of all given τ-transitions. This is not the case, however, with the frame of GifToJpg which shows an internal choice after receiving gif at port j. An implementation will either proceed and send a jpg back via j or cancel immediately and wait for another gif. In case a message gif is received in between, the current jpg is dismissed.

In Chap. 2 we introduced behaviours for the components GZip and GifToJpg (Fig. 2.3) which indeed are correct behaviours w.r.t. the frame specifications given in Fig. 6.2d and Fig. 6.2f. The behaviour of GZip reifies the output transition $(z_1, \text{z.stop}, z_2)$ and disregards the possibility to send out cont acknowledgements, while the behaviour of GifToJpg merely provides named labels process and cancel for the anonymous internal labels given by the frame specification.

Still missing is the above mentioned component to incorporate the given tool components in such a way that the resulting composition is a valid implementation of the compressing proxy component. Figure 6.3 shows the frame of an adaptor component which provides such an integration in a simple way. We assume ports of the component given according to the labels used for the specification of its frame. The component passes either textual data to the zip component, or graphical data to the JPG conversion tool. After compression is finished, the result is sent further downstream and the adaptor is ready to receive further textual or graphical data. The messages communicated in between mirror the frame of GZip, or use the services provided by the frame of GifToJpg in a rather direct way respectively.

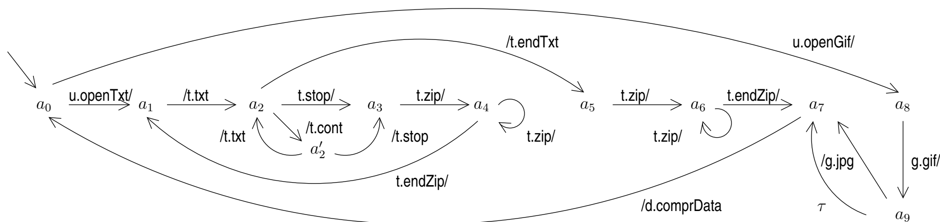


FIGURE 6.3. Frame specification of component Adaptor

In order to complete the specification of the compressing proxy component we further need to decide on the communication timing between the given components, that is we need to decide on the kind of connectors. Since the protocols were designed with a

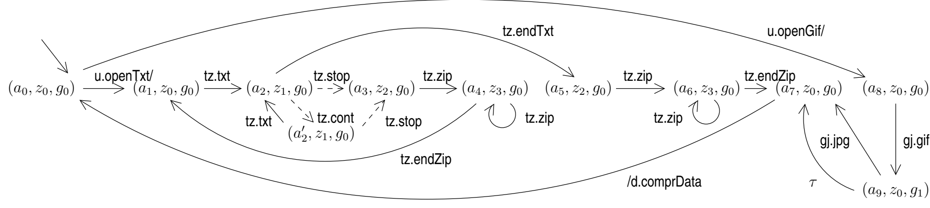


FIGURE 6.4. Frame composition using synchronous connectors

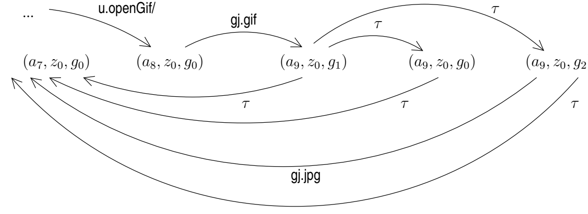


FIGURE 6.5. Composition of Adaptor and GifToJpg

rendezvous-like mechanism in mind, we set up a static structure with synchronous connectors. For the remaining part we assume static elements as given by the original design in Chap. 2 (Fig. 2.2). Then, the composition of frames and synchronous connectors yields an input-persistent I/O-transition system (PIO) as depicted in Fig. 6.4. Here and in the following the component names do not contribute to the discussion and thus we simplify the representation, omitting the names of component declarations for better readability. Moreover the composition was simplified with regard to τ-transitions in the synchronisation part between Adaptor and GifToJpg (transitions (a_8, z_0, g_0) to (a_7, z_0, g_0)). The original product yields an interleaving as depicted in Fig. 6.5 which is observational equivalent to the simplified representation in Fig. 6.4. Since all transitions in between receiving messages upstream and sending compressed data further downstream are internal transitions, it is easily verified that the resulting frame assembly indeed is a refinement of the initial requirements. More formally, we have

$$\mathit{frm}(\langle \mathcal{C}; \mathcal{K} \rangle)\rho \sqsubseteq^{\mathrm{bb}} \mathit{frm}(\mathsf{CompressingProxy}),$$

where $\mathcal{C} = \{$adapt:Adaptor, gzip:GZip, gif:GifToJpg$\}$, $\mathcal{K} = \{$tz:(t:TxtCompr, z:Zip), gj: (g: GifCompr, j:ToJpg)$\}$ and ρ is a relay relabelling which maps the port names u and d used for Adaptor to the port names l and r used for the specification of CompressingProxy. Note that the mentioned minimisation w.r.t. observational equivalence does not touch the validity of refinement. Moreover, compatibility analysis is not affected either, as long as weak or ultra-weak output compatibility is concerned. This is not the case if we aim at proofing strong output compatibility, since observational equivalence allows to add τ-transitions as long as the weak bisimulation relation is not invalidated and thus strong compatibility is not preserved.

2.2. Communication-Safety. Frame analysis is one of the advantages of an approach to component-based development that explicitly distinguishes component specifications and implementations. In our model such a distinction manifests in the difference between frames and behaviours of a component. The frame is usually a more abstract description, representing a whole class of possibly (correct) implementations, and thus for any property which is known to be preserved by the implementation relation between frames and behaviours it suffices to analyse component frames instead of behaviours. As a representative example Prop. 6.5 allows to conclude weak and ultra-weak comm-safety for compositions of correct component behaviours.

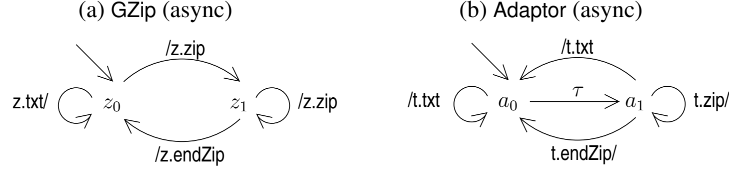


FIGURE 6.6. GZip and Adaptor for buffered communication

There are several possibilities to analyse comm-safety. Consider, for instance, the frame composition in Fig. 6.4. First, since the design of the adaptor component followed the given frame specifications of the tool components rather closely, we may expect pairwise weak output compatibility to hold between the given frames. In this case Prop. 4.37 would be applicable to conclude weak comm-safety of the given frame composition. Indeed, an analysis for Adaptor and GZip even yields strong output compatibility, that is we have

$$frm(\mathsf{Adaptor})\sigma \leftrightarrow_s frm(\mathsf{GZip})\sigma \text{ and hence } frm(\mathsf{Adaptor})\sigma \leftrightarrow_u frm(\mathsf{GZip})\sigma,$$

where σ is the synchronous relabelling w.r.t. the connector tz:(t:TxtCompr, z:Zip) (again note that we left out the component names for better readability). Unfortunately this is not the case for the frames of Adaptor and GifToJpg. Even worse, the frames are not output compatible at all, due to the transition $(g_2, \text{j.jpg}, g_0)$ (Fig. 6.2f) being globally reachable in state (a_9, z_0, g_1) (Fig. 6.4). In order to fix this defect, we need to add appropriate reception transitions for the jpg messages sent by GifToJpg. Adding two reception loops to $frm(\mathsf{Adaptor})$, one in state a_7 and one in state a_0, yields weakly output compatible frames. Now, since $frm(\mathsf{GZip})$ and $frm(\mathsf{GifToJpg})$ are obviously output compatible, Prop. 4.37 is applicable and it follows that the composition in Fig. 6.4 is weakly comm-safe. Alternatively it would also suffice to add only the reception loop in state a_0. In this case, we obtain ultra-weak output compatibility between Adaptor and GifToJpg which is, however, not strong enough to conclude comm-safety from pairwise analysis. Instead, an incremental analysis using Prop. 4.40 is necessary to show ultra-weak comm-safety for the given composition. The difference between pairwise and incremental analysis draws a clear border for the applicability of port-based techniques for verification (cf. Sect. 4) to which the feasibility of pairwise analysis is of paramount importance.

We did not yet consider frame analysis with asynchronous connectors. Indeed, frame design of port-based components is in general strongly influenced by assumptions on the underlying communication mechanism. There might be frame specifications that are well-suited for both kinds of communication timing, but there might be protocols too that fit either to the synchronous but not to the asynchronous case or vice versa. For example, the adaptor component designed above yields a rather complicated behaviour in case an asynchronous connector is employed between the components Adaptor and GZip. With buffered communication in mind simpler designs such as, for example, the frames as depicted in Fig. 6.6 could be used. The transition systems are not ultra-weak output compatible due to the output transition $(a_1, \text{t.txt}, a_0)$ in the adaptor frame on the right-hand side. The global control state (z_1, a_1) is reachable, but the frame of GZip on the left-hand side provides to receive txt only after the shared output transition $(z_1, \text{z.endZip}, z_0)$. For ultra-weak compatibility, such a transition would need to be either internal or non-shared output.

The frame specifications of Fig. 6.6 combined with an asynchronous connector yield a global transition system which is infinite-state due to the possibility on both sides of the communication to send messages without ever receiving one. The adaptor component may send the message txt and GZip may send the message zip arbitrary often. In order to confirm asynchronous compatibility for the given specifications one needs to apply either criteria

which are applicable to the given finite-state control automata (frames) or, for instance, symbolic approaches such as the QDD approach described in Chap. 5.[2]

Once weak (ultra-weak) comm-safety is established for an assembly of component frames with synchronous or asynchronous connectors, Prop. 6.5 allows to conclude weak (ultra-weak) comm-safety for the implementation of the system as long as components are correctly implemented. Remember that our implementation relation (cf. Def. 6.2) is independent from the particular kind of connectors and hence works for systems with both, synchronous and asynchronous communication. Moreover it is important to note that this holds for all behaviours which are correct w.r.t. the frame of a given component. For example using an implementation of the component GZip with a behaviour that neither sends cont nor stop is perfectly acceptable with regard to $frm(\mathsf{GZip})$ in Fig. 6.2d, and the composition of the system on behavioural level is still comm-safe due to comm-safety on the frame level of the system.

3. Supporting Component-Based Development

In this section we show that our approach indeed supports characteristic features of component-based development. First, we examine support for top-down and bottom-up design approaches.

3.1. Top-Down and Bottom-Up Design. Top-down design decomposes a single specification into a number of separated specifications whose composition is either equivalent or a refinement of the original global specification. Then, if we use implementations for the composition's parts, we are allowed to conclude that the composition of the implementations is a valid implementation of the original single specification. Bottom-up approaches start with detailed single components which are composed to create an implementation of some given specification. Under the assumption of correct component implementations our approach supports both, top-down and bottom-up approaches to component-based design.

Theorem 6.8 (Bottom-up) *Let $a \in \mathrm{Asm}$ such that for all $c : C \in cmps(a)$, $beh(c : C) \in [\![frm(c : C)]\!]$. Let $H^{\triangleright} = \bigcup\{l^{\triangleright} \mid l^{\triangleright} \in T_{frm(a)}\}$. Then $beh(a) \setminus H^{\triangleright} \sqsubseteq^{\mathrm{gb}} frm(a) \setminus H^{\triangleright}$ and $beh(a) \sqsubseteq^{\mathrm{bb}} frm(a)$.*

PROOF. The first claim holds by Lem. 6.7 and then, $beh(a) \sqsubseteq^{\mathrm{bb}} frm(a)$ follows from Prop. 4.43 (1) and the invariance of blackbox refinement w.r.t. hiding of internal labels (cf. Lem. 4.5). □

Theorem 6.9 (Top-down) *Let $C \in \mathrm{Cmp}$ and let $a \in \mathrm{Asm}$ such that for all $d : D \in cmps(a)$, $beh(d : D) \in [\![frm(d : D)]\!]$. If $frm(a) \sqsubseteq^{\mathrm{bb}} frm(C)$ then $beh(a) \in [\![frm(C)]\!]$.*

PROOF. By Thm. 6.8, transitivity of blackbox refinement, and Def. 6.2 of correct implementations. □

Example 6.10 (Top-down/Bottom-up) *The compressing proxy system above was developed using a mixture of top-down and bottom-up steps. Given the frame of the compressing proxy component, we seeked to identify existing components whose behaviour composition can be used to implement the given component. Thus, the first design step followed a bottom-up approach. Once appropriate components had been identified with GZip and GifToJpg, however, the direction switched and we considered their frame specifications and designed an adaptor component, aiming at establishing blackbox refinement between the composition of the frame specifications of all three components and the frame specification as given for the compressing proxy component.*

[2]The examples have been checked by hand.

3.2. Evolution in Hierarchical Systems. Besides the general support of component-based design approaches, any formal component model should also support modular substitutability based on its refinement and implementation relations. We consider the notion of "component-wise evolution" [**dAH01b**]. Evolution aims at replacing an implementation without changing the frame of the original component. In this case it should suffice to check only correctness w.r.t. the frame of the original component in order to conclude the correctness of an evolved assembly. For the presentation of subsequent claims we will use the notations for type substitution in assemblies and composite components defined in Chap. 2 (Def. 2.22/2.23).

Proposition 6.11 (Evolution) *Let* $CC \in \mathrm{CCmp}$. *Let* $c : C \in cmps(asm(CC))$ *and let* $C' \in \mathrm{Cmp}$ *with* $ports(C) = ports(C')$. *If* $beh(CC) \in [\![frm(CC)]\!]$ *and* $beh(C') \in [\![frm(C)]\!]$ *then* $beh(CC[c : C \mapsto C']) \in [\![frm(CC)]\!]$.

PROOF. By Def. 6.2 of correct implementations we have $pio(beh(C')) \sqsubseteq^{\mathrm{bb}} frm(C)$. By Prop. 4.10 it follows that $pio(beh(asm(CC)[c : C \mapsto C'])) \sqsubseteq^{\mathrm{bb}} pio(beh(asm(CC)))$. Now, the claim follows by transitivity of blackbox refinement (cf. Prop. 4.8) and the definition of correct implementations. □

Example 6.12 (Evolution) *Replacing the component GZip by a different component, say GZip' which does not send stop messages is still correct w.r.t.* $frm($*GZip*$)$ *in 6.2d, i.e.* $beh($*GZip'*$) \in [\![frm($*GZip*$)]\!]$ *and thus, by Prop. 6.11, we may replace GZip by GZip' and still have a correct implementation of* $frm($*CompressingProxy*$)$.

The following proposition shows the support of evolution for hierarchical component-based systems of depth two.[3] The proposition can be used to show support of evolution in arbitrary hierarchical systems by induction on the depth of the hierarchy.

Proposition 6.13 (Hierarchy) *Let* $CC^T \in \mathrm{CCmp}$ *such that* $beh(CC) \in [\![frm(CC)]\!]$ *and let* $cc : CC \in cmps(asm(CC^T))$. *Let* $c : C \in cmps(asm(CC))$ *and* $C' \in \mathrm{Cmp}$ *with* $ports(C) = ports(C')$. *If* $beh(C') \in [\![frm(C)]\!]$, *then*

$beh(CC^T[cc : CC \mapsto CC']) \in [\![frm(CC^T)]\!]$,
where $CC' = CC[c : C \mapsto C']$.

PROOF. Using Prop. 6.11 we conclude from $beh(C') \in [\![frm(C)]\!]$ that CC' with $CC' = CC[c : C \mapsto C']$ is still correct, that is $beh(CC') \in [\![frm(CC)]\!]$. Now, for $cc : CC \in cmps(asm(CC^T))$ it follows again by Prop. 6.11 that $beh(CC^T[cc : CC \mapsto CC']) \in [\![frm(CC^T)]\!]$. □

3.3. Weak- and Ultra-Weak Communication-Safety. So far we have discussed general support for top-down and bottom-up design approaches as well as general support for evolving (hierarchical) systems. Next we show that frame analysis for ultra-weak comm-safety carries over to frame refinements in top-down design approaches. Moreover, ultra-weak comm-safety is preserved under substitution of correct component behaviours for evolving (hierarchical) systems. The succeeding claims also hold for weak comm-safety. Strong compatibility, however, is not preserved by blackbox-refinement and hence strong comm-safety does not carry over. In order to get analogous results for strong comm-safety, a strong refinement relation must be applied that does not allow for the insertion and removal of internal transitions.

Proposition 6.14 (Comm-safe top-down design) *Let* $a \in asm$ *such that for all* $d : D \in cmps(a)$, $beh(d : D) \in [\![frm(d : D)]\!]$. *Let* $c : C \in cmps(asm(a))$ *and let* $C' \in \mathrm{Cmp}$ *with*

[3] The statement is similar to the claim on behaviour preservation for arbitrary nested hierarchies in the context of behaviour protocols [**PV02**, Thm. 3.4.4] (behaviour protocols are used as a formal model for component behaviours in SOFA).

$ports(C) = ports(C')$. If $frm(a)$ is weakly (ultra-weakly) comm-safe and $frm(C') \sqsubseteq^{\mathrm{bb}} frm(C)$, then $frm(a[c : C \mapsto C'])$ is weakly (ultra-weakly) comm-safe.

PROOF. The claim is a consequence of Prop. 4.15 (Prop. 4.19 resp.) along an expansion of definitions similar to the proofs for the support of bottom-up and top-down design approaches. □

It is not strictly necessary to refine frames in order to preserve weak or ultra-weak comm-safety. By component-wise evolution we may also replace components using a different (but correct) implementation while leaving the frame specification untouched.

Proposition 6.15 (Comm-safe evolution) *Let $a \in asm$ such that for all $d : D \in cmps(a)$, $beh(d : D) \in [\![frm(d : D)]\!]$. Let $c : C \in cmps(asm(a))$ and let $C' \in \mathrm{Cmp}$ with $ports(C) = ports(C')$. If $frm(a)$ is weakly (ultra-weakly) comm-safe and $beh(C') \in [\![frm(C)]\!]$, then $beh(a[c : C \mapsto C'])$ is weakly (ultra-weakly) comm-safe.*

PROOF. By Prop. 6.14 and definition of correct implementations (Def. 6.2) it follows that $frm(a[c : C \mapsto C'])$ is weakly (ultra-weakly) comm-safe. Hence $beh(a[c : C \mapsto C'])$ is weakly (ultra-weakly) comm-safe by Prop. 6.5. □

Example 6.16 (Comm-safe top-down design and evolution) *Consider the compressing proxy example and assume $frm(\langle \mathcal{C}; \mathcal{K} \rangle)$ with $\mathcal{C} = \{$adapt:Adaptor, gzip:GZip, gif:GifToJpg$\}$ and $\mathcal{K} = \{$tz:(t:TxtCompr, z:Zip),gj: (g: GifCompr, j:ToJpg)$\}$ is comm-safe (e.g. with frame specifications as depicted in Fig. 6.2d, Fig. 6.2f and Fig. 6.3, and synchronous connectors $\mathcal{K} \subseteq \mathrm{SynConn}$). Moreover, assume that*

(i) one of the frames is refined, or
(ii) one of the implementations is replaced.

Then, (i) can be used to illustrate the preservation of comm-safety in top-down designs and (ii) to illustrate the preservation of comm-safety in evolving systems. For instance, assume GZip' is a component which refines the frame of GZip to a frame which does not support cont acknowledgements. Then, we may conclude along Prop. 6.14 that the frame composition using the new component is still comm-safe. More formally we may conclude comm-safety of $frm(\langle \mathcal{C}; \mathcal{K} \rangle$[gzip:GZip $\mapsto$ GZip']$)$. For (ii) we replace a component using the original frame specification but a different implementation. If the new component is correct, then we may use Prop. 6.15 to conclude that comm-safety on the behavioural level still holds. More formally, with a component GZip', showing a behaviour such that $beh($GZip'$) \in [\![frm($GZip$)]\!]$ and $ports($GZip$) = ports($GZip'$)$ we may conclude comm-safety of $beh(\langle \mathcal{C}; \mathcal{K} \rangle$[gzip:GZip $\mapsto$ GZip']$)$.

Finally, we extend our claim to show support for hierarchies of depth two. The proposition can be used to show support for arbitrary hierarchies by induction on the depth of the hierarchy.

Proposition 6.17 (Comm-safe hierarchy) *Let $CC \in \mathrm{CCmp}$ such that $beh(asm(CC)) \in [\![frm(CC)]\!]$. Let $c : C \in cmps(asm(CC))$ and let $C' \in \mathrm{Cmp}$ with $ports(C) = ports(C')$. If $frm(asm(CC^T))$ is weakly (ultra-weakly) comm-safe and $beh(C') \in [\![frm(C)]\!]$ then*

$beh(asm(CC^T)[cc : CC \mapsto CC'])$ is weakly (ultra-weakly) comm-safe,
where $CC' = CC[c : C \mapsto C']$.

PROOF. By Prop. 6.13 we know $frm(asm(CC^T)[cc : CC \mapsto CC']) \sqsubseteq frm(CC^T)$. Since $frm(CC^T)$ is weakly (ultra-weakly) comm-safe and $\sqsubseteq$ preserves weak (ultra-weak) comm-safety (Prop. 4.15, Prop. 4.19), it follows that $frm(asm(CC^T)[cc : CC \mapsto CC'])$ is weakly (ultra-weakly) comm-safe. Finally, since weak and ultra-weak comm-safety do not distinguish internal labels and anonymous internal actions labelled τ, it follows that $beh(asm(CC^T)[cc : CC \mapsto CC'])$ is weakly (ultra-weakly) comm-safe. □

4. Port-Based Frame Verification of Communication-Safety

The general idea of port-based verification is to use ports as partial views to components and analyse instead of component interaction only the interaction which is visible at the particular ports. In our component model this means to consider the connectors of an assembly. The pairwise analysis is then used to conclude global properties of the underlying assembly behaviour. Of course such an approach is only applicable if the property under analysis indeed is a consequence of pairwise considerations. As an example consider our notions of output compatibility in n-ary compositions, i.e., strong, weak, and ultra-weak comm-safety. While weak comm-safety follows from pairwise analysis (Prop. 4.37), ultra-weak comm-safety does not (Ex. 4.39). Therefore we will focus on weak output compatibility in the following.

In contrast to our approach using ports for a more efficient check between component behaviours in Chap. 3, we assume the mentioned port view to be explicitely given, i.e. we assume an explicitely specified port protocol. Note that, analogous to frame specifications, τ-transitions are still allowed but internal actions should not be visible any more.

Definition 6.18 (Port protocol) *The protocol of a port* $P \in \mathrm{Port}$ *is an input-persistent I/O-transition system* $prot(P) = ((I, O, T), S, s_0, \Delta, \Pi)$ *with* $I = msg(prv(P))$*,* $O = msg(req(P))$ *and* $T = \emptyset$*. The protocol of a port declaration is given by* $prot(p : P) = p.prot(P)$*.*

Figure 6.7 shows the corresponding extension of our metamodel. Ports are equipped with an optional protocol that may be used to specify partial views with regard to the component's frame. Such a protocol may support verification of global properties by pairwise checks between port protocols of the components. As an example we consider weak comm-safety that is known to support pairwise verification on the level of frame specifications (Prop. 4.35).

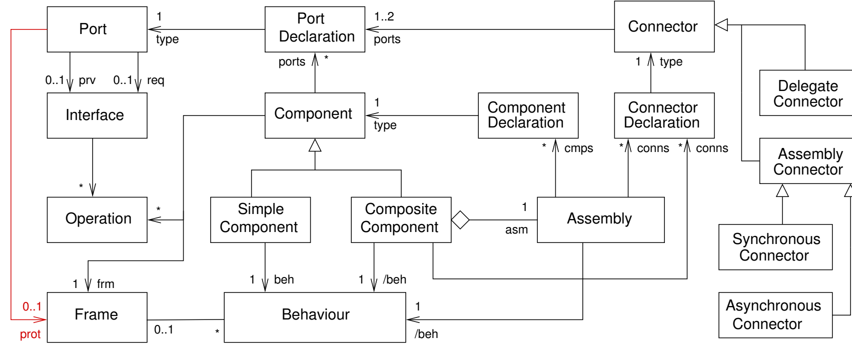


FIGURE 6.7. Metamodel with frames and port protocols

The analysis is based on the following notion of connector compatibility.

Definition 6.19 (Comm-safe connectors) *An assembly connector* $k : K \in \mathrm{ConnDcl}$ *with* $ports(K) = \{p : P, q : Q\}$ *is* weakly comm-safe *if* $K \in \mathrm{SynConn}$ *implies* $prot(P) \leftrightarrow_w prot(Q)$*, and* $K \in \mathrm{AsynConn}$ *implies* $\Omega(prot(P)) \leftrightarrow_w \Omega(prot(Q))$*.*

Communication-safety in assemblies relates to the frame specifications of components. Therefore, we need a formal relation between the frame specification and the port protocols of a component. We will use blackbox refinement for this purpose since this is our weakest relation between PIOs that is known to preserve weak output compatibility for both, synchronous and asynchronous communication. We first consider the case of two components.

Lemma 6.20 (Port-based weak comm-safety) *Let* $a = \langle c : C[p : P], d : D[q : Q]; k : K\rangle$ *be an assembly such that* $ports(K) = \{c.p : P, d.q : Q\}$*. If* $frm(C) \sqsubseteq^{\mathrm{bb}} prot(p : P)$*,* $frm(D) \sqsubseteq^{\mathrm{bb}} prot(q : Q)$ *and* $k : K$ *is weakly comm-safe then* $beh(a)$ *is weakly comm-safe.*

PROOF. We distinguish $k : K$ being either synchronous or asynchronous. If $k : K$ is a synchronous connector then $prot(P) \leftrightarrow_w prot(Q)$. Since compatibility is preserved by prefix relabelling we also have $k.prot(P) \leftrightarrow_w k.prot(Q)$. By Prop. 4.19 for $frm(C)\sigma \sqsubseteq^{\mathrm{bb}} k.prot(p : P)$ it follows that $frm(C)\sigma \leftrightarrow_w k.prot(Q)$ where σ is the synchronous relabelling $\sigma_{(\{c.p,d.q\},k)}$. By a symmetric reasoning with port Q and component D it follows that $frm(c : C)\sigma \leftrightarrow_w frm(d : D)\sigma$ and hence $beh(a)$ is weakly comm-safe.
If $k : K$ is an asynchronous connector it holds that $\Omega(prot(P)) \leftrightarrow_w \Omega(prot(Q))$ and also $\Omega(k.prot(P)) \leftrightarrow_w \Omega(k.prot(Q))$. Denote the asynchronous IOL of $k.prot(P)$ by L_P. Then it follows by Prop. 4.32 for $frm(C)\alpha \sqsubseteq^{\mathrm{bb}} k.prot(p : P)$ that $\Omega(frm(c : C)\alpha, L_P) \leftrightarrow_w \Omega(k.prot(Q))$ where α is the asynchronous relabelling $\alpha_{(\{c.p,d.q\},k)}$. Hence, by a symmetric reasoning, $\Omega(frm(c : C)\alpha, L_P) \leftrightarrow_w \Omega(frm(d : D)\alpha, L_Q)$ which means that $beh(a)$ is weakly comm-safe. □

For the generalisation to arbitrary assemblies we consider only assemblies which are either completely synchronous or completely asynchronous. An assembly a is *completely synchronous* if $acs(a) = \emptyset$; it is *completely asynchronous* if $(scs \setminus ucs) = \emptyset$, that is, the only synchronous connectors are unary. The restriction to a homogenous communication timing is due to the corresponding restriction on the level of PIOs in Sect. 3 of Chap. 4. In order to verify heterogenous assemblies, components that communicate synchronously may be gathered and replaced by an equivalent component such that only components communicating via asynchronous communication remain.

Corollary 6.21 (Weak comm-safety) *Let* a *be an assembly which is either completely synchronous or asynchronous. If for all assembly connectors* $k : (c.p : P, d.q : Q) \in conns(a)$ *it holds that* $frm(cmp(c.p : P)) \sqsubseteq^{\mathrm{bb}} prot(p : P)$ *and* $frm(cmp(d.q : Q)) \sqsubseteq^{\mathrm{bb}} prot(q : Q)$ *and* $k : K$ *is weakly comm-safe then* $beh(a)$ *is weakly comm-safe.*

PROOF. By Lem. 6.20 and Prop. 4.37 for synchronous assemblies. Since buffered PIOs are PIOs, Prop. 4.37 also applies to the case of asynchronous connectors. □

The refinement requirements for the component frames with regard to the port protocols are rather strong assumptions. As the complete frame specification needs to be in relation, any input or output of the particular component needs to be taken into account. In particular this means that all of the connected ports of a component are equipped with the same alphabet. Even if this might be handled along mechanisms such as alphabet extension as applied in process algebras, it remains the requirement that the frame specification integrates the port protocols in an independent way. That is, the communication at different ports is not interleaved at all which is obviously quite a strong requirement for the frame specification of a component.

Therefore, it would be important to weaken the assumptions in such a way that it is not necessary to take the complete frame specification into account. For instance, we may hide all output transitions that do not involve the currently considered port. Since we usually assume output compatibility, this assumption should not affect the correctness of the global conclusion. More precisely, denote the output labels of a component frame $frm(C)$ by O_C and assume $prot(p : P)$ to be the protocol of a port of C. Let $H^{\mathrm{out}}_{\neg p} = O_C \setminus \{p.m \mid m \in msg(req(P))\}$. Then, a possibly weaker assumption would require $frm(C)/H^{\mathrm{out}}_{\neg p} \sqsubseteq^{\mathrm{bb}} prot(p : P)$ allowing to conclude analogously to the reasoning above that $frm(C)/H^{\mathrm{out}}_{\neg p}$ and $frm(D)/H^{\mathrm{out}}_{\neg q}$ are weakly compatible. However, weak compatibility with such a hiding corresponds to ultra-weak compatibility and ultra-weak compatibility does not allow for pairwise analysis (Ex. 4.39). Hence, the result from the

binary case does not carry over to n-ary compositions, i.e., to weak comm-safety of assemblies.

On the contrary hiding inputs is not backed by any compatibility assumption. Thus we do not expect to find weaker assumptions for the general case of assemblies with an arbitrary topology. We believe, however, that an analysis of acyclic assemblies similar to our approach for the detection of neutral component behaviours and a syntactical reduction of assemblies in Chap. 3 also works for a port-based verification of weak comm-safety with such a weaker assumption. Hence an integration of the reduction approach of Chap. 3 with an analysis of comm-safety could be an interesting issue for future work.

5. Related Work

This chapter completes the formal behavioural model of Chap. 2 with a more abstract specification layer using component frame specifications. The resulting formal software component model was shown to support fundamental characteristics of component-based development. Such a support is formally also considered by SOFA 2.0 [**PV02**] and by Component Interaction Automata [**CVZ06**]. The former discusses support for hierarchical composition within arbitrarily nested hierarchies, and the latter develops a generic notion of equivalence that probably could be used to prove support of top-down and bottom-up design similar to our approach. Moreover, their notion of independent implementability seems to be similar to our notion of evolution in hierarchical systems.

Component frames provide a specification of a component's behaviour that is usually more abstract compared to the formal representation of its implementations. The difference between specification and implementation is also examined by de Alfaro and Henzinger in [**dAH01b**]. Their framework sheds some light on the difference and the interrelation between component implementation and specification, called interfaces in [**dAH01b**], in the context of component-based design. While the representation of a component's implementation is supposed to describe what the component actually does, an interface description should describe how the component can be used. Formally distinguished component and interface algebras are used for an abstract representation of component implementations and interfaces. Both algebras are equipped with a form of composition and hierarchy (refinement). Then, an interface algebra together with an implementation relation that relates to a component algebra is called an interface theory. In our terminology, this would be called a formal software component model while we call the interface algebra an interface theory. Their study of support for top-down and bottom-up design, and evolution is conceptually closely related. According to [**dAH01b**], interfaces should support top-down design, components should support bottom-up verification, and support for evolution (implementation substitutability) is support for a task that replaces a component (type) without observable effect on the original composition.

CHAPTER 7

UML2 – Applied Features and Extensions

1. Static Structure of Components and Assemblies 113
2. Component Behaviour and UML State Machines 117
3. Frame Specifications and UML Protocol State Machines 119
4. From State Machines to Transition Systems 120

Our concrete specifications comprise UML2 class and composite structure diagrams for the description of the static structure of components, ports, interfaces, and especially of composite components and their assemblies.[1] We use UML2 state machines to specify component behaviours and protocol state machines for component frames. Within this chapter we summarise applied UML2 features and detail on the meaning of additional notations and stereotypes. We relate static structure elements with our formal model and discuss an informal pattern-based translation from state machines to transition systems. Based on this translation we define the relation between UML state machines and protocol state machines and component behaviours and frames as considered in Chap. 6. The given translation allows to apply the theory for IOTSs and PIOs developed in Chap. 4 as a formal underpinning for the definition of a UML-based theory of refinement (protocol conformance) and compatibility. In combination with a corresponding extension of a UML modelling tool this would provide a convenient formal tool for UML-based design and verification of port-based component systems.

1. Static Structure of Components and Assemblies

Table 7.1 provides an overview of the elements for static structure specifications of our component model and their correspondence to the UML. In order to provide a link to our formal model, we have included selected formal definitions. In the following, however, we elaborate each of the mentioned elements only conceptually and refer to Chap. 2 for details on the formal notation.

Port-based components have their correspondence with UML components that are encapsulated by UML ports. We use either behaviour or delegation ports; the former in case of simple components and the latter in case of composite components (see [**RJB05**, Fig. 14-216] for a general illustration of the difference between behaviour and delegation ports). UML ports are equipped with required and provided interfaces which are used to specify operations that are available or needed at the particular port. The interfaces in our component model specify signal receptions only. For a derivation of signal types we use the following convention. Consider the interface in Fig. 7.1 taken from the CoCoME example below.
The interface specifies two operation signatures within a compartment labelled by the UML stereotype «signal». In this way, signal receptions are specified which is just a statement that the implementing "classifier is prepared to react to the receipt of a signal" [**OMG09**, Sect. p. 449]. Instead of an explicit type specification of the associated signal we implicitly

[1] The difference between class diagrams and composite structure diagrams is not strict. As mentioned in [**RJB05**]: "*There is no rigid line between a composite structure diagram and a general class diagram*".

TABLE 7.1. Component model and UML2 modelling elements

Concept	Formal Model (selected elements)	UML2 modelling element
Port	$P \in \text{Port}, p : P \in \text{PortDcl}$	Behaviour/Delegate ports
Interface	$req(P), prv(P) \in \text{If}$	Interface with «signal»
Message	$msg(req(P)), msg(prv(P)) \subseteq \text{Msg}$	Signal object
Component	$C \in \text{Cmp}, ports : \text{Cmp} \to \wp\text{PortDcl}$	Component with ports
Simple component	$\text{SCmp} \subseteq \text{Cmp}$	Component with ports
Connector	$K \in \text{Conn}, ports : \text{Conn} \to \wp\text{PortDcl}$	Connector
Assembly conn.	$\text{AsmConn} \subseteq \text{Conn}, req/prv$ vs prv/req	Assembly conn.
Delegate conn.	$\text{DlgConn} \subseteq \text{Conn}, req/prv$ vs req/prv	Delegate conn.
Synchronous conn.	$\text{SynConn} \subseteq \text{Conn}$	Asm. conn. with «sync»
Asynchronous conn.	$\text{AsynConn} \subseteq \text{AsmConn}$	Asm. conn. with «async»
Assembly	$a \in \text{Asm}, cmps : \text{Asm} \to \wp\text{CmpDcl}$, $conns : \text{Asm} \to \wp\text{ConnDcl}$	Internal structure
Composite component	$\text{CCmp} \subseteq \text{Cmp}, asm : \text{CCmp} \to \text{Asm}$, $conns : \text{CCmp} \to \wp\text{ConnDcl}$	Structured classifier

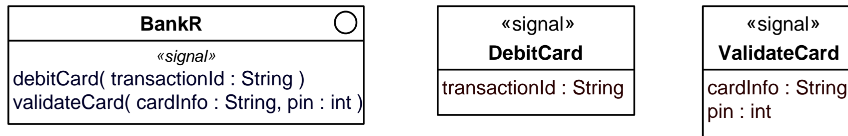


FIGURE 7.1. Interfaces and signal types

assume corresponding signal type specifications DebitCard and ValidateCard as illustrated in the given figure. The type names are derived from the operation names and the attributes correspond to the given parameters. Our concept of messages relates to an object of the respective signal type, thus there are as many messages as instantiations of the given type exist. Since we do not consider data states on the formal level of component behaviours, however, a message when used as a label in a transition system usually consists of a name only. The translation from received or sent UML signals to labels of transition systems is detailed below.

The difference between simple and composite components has its counterpart in using UML components with ports for simple components and UML components as structured classifier for the representation of composite components. A structured classifier contains parts and connectors that realise its behaviour. The ports of a structured classifier define its external interaction points and are connected with the open ports of the internal parts by UML delegate connectors. Inside the structured classifier, UML assembly connectors are used. Our concept of assembly corresponds to the internal structure of the classifier *without* delegate connectors and thus does not have a true UML counterpart in the sense that assemblies are first-class citizen with a possibility to be specified stand-alone. In UML, parts and their connectors need always to be set into the context of a structured classifier.

The parts, connectors and ports are in fact declarations that specify name, type and multiplicity of the particular entity. By the "part concept" UML introduces the possibility to reuse types, in particular component types in different contexts. This reuse possibility is important for component-based development as component reuse is at the core of CBSE (cf. Chap. 1, Sect. 1). For connector declarations we often specify the name of assembly connectors only and leave other information unspecified or implicitly specified by derivation from the connected port types. Connector multiplicities are in principle independent

from port multiplicities. Though, we restrict our connectors to singleton types whose multiplicity 1 is not explicitly shown in the diagrams. Moreover, we do not consider group communication and hence allow only for binary connectors.

As an example consider the compressing proxy component introduced in Chap. 2 whose static structure specification is for convenience repeated here in Fig. 7.2. The composite component type CompressingProxy consists of an assembly of three simple components with types Adaptor, GZip, and GifToJpg introduced by the respective component declarations adapt:Adaptor, gzip:GZip, and gifToJpg:GifToJpg. The simple components as well as the composite component show port declarations such as t:TxtCompr or l:UpStream.

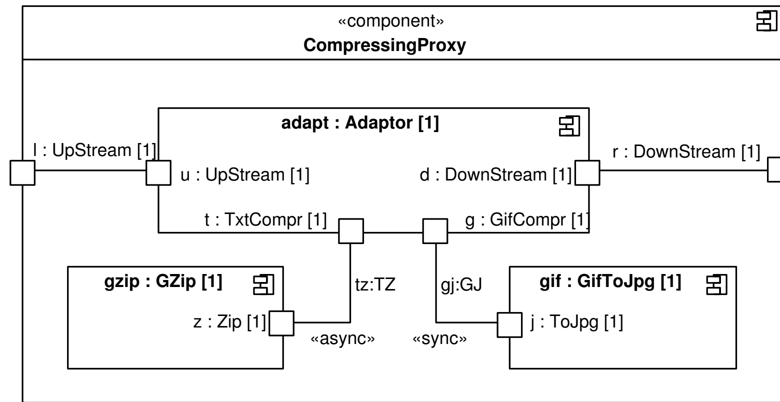


FIGURE 7.2. Using UML2 for the specification of composite components

In the given example we use only singleton types for both, components and ports. The components are connected with assembly connectors named tz and gj. The anonymous delegate connectors link the open port declarations u:UpStream and d:DownStream of the assembly with the port declarations l:UpStream and r:DownStream of the composite component type CompressingProxy.

We introduce three stereotypes for specific kinds of connectors. Stereotypes «sync» and «async» specify the communication timing and the stereotype «seq-adapter» an adaption pattern for connections between ports with different multiplicities. Figure. 7.3 shows an example for the specification of an asynchronous and an adapter connector. The communication timing relates to the message exchange between components. With a synchronous connector a rendezvous (handshake) mechanism is used and with an asynchronous connector FIFO buffers are used, one for each communication direction. The precise meaning of «sync» and «async» is formally given by the definition of buffering connector behaviours in Chap. 2, Def. 2.21. and their integration with component behaviours in Def. 2.22.

The stereotype «seq-adapter» is used to bridge communication links between ports that are specified with different multiplicities. Consider, for instance the specification of the CoCoME component CashDeskLine in Fig. 8.2a below. There might be several instances of type CashDesk according to the multiplicity 1..* of the component's declaration. With a sequential adapter connector we think of an additional component that provides as many ports as there are instances of CashDesk and exactly one port to face the communication with the bank. Figure. 7.3 illustrates the corresponding expansion of the static structure for the connector between the bank port of possibly several cashDesks and the bank port of the composite component CashDeskLine. The behaviour of a «seq-adapter» connector is supposed to realise the communication between n cashdesks and one bank by *sequential* (synchronised) arrangement of multiple requests of different cash desk components.

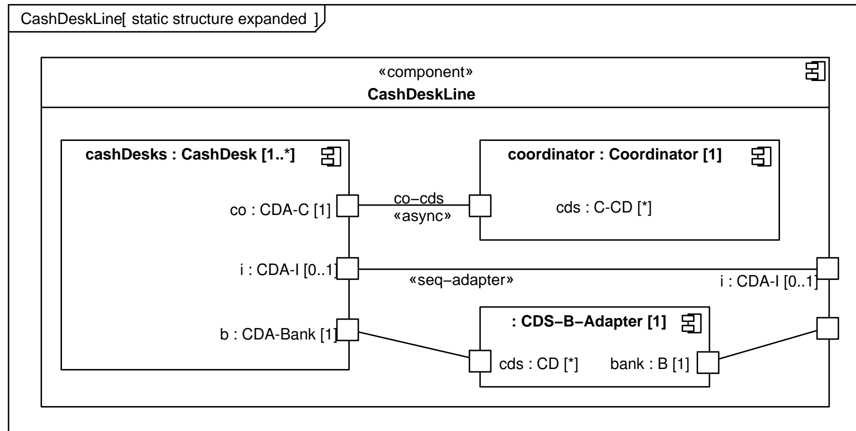


FIGURE 7.3. Sequential adapter in component CashDeskLine

Finally, we allow for the static modelling of attributes and operations for components and ports. The former provides a convenient way to specify data-dependent component behaviour and the latter is used to declare internal methods that help to illustrate the intended meaning of internal actions. Since we do not consider data states on the formal behavioural level of our component model, the data-related specifications need to be abstracted or instantiated for the translation to I/O-transition systems. For this reason we did not include these concepts in our metamodel in Chap. 2 (Fig. 2.1). The precise relation to the UML is analogous to the relation between the Java/A component model and UML as defined in [**KJH$^+$08a**, Fig. 1]. Note that component, port and connector declarations are represented by metaclasses with the suffix Property instead of Declaration in [**KJH$^+$08a**].

Multiplicities of Ports and Components. The formal part of our component model is not set up to cope with issues of dynamic reconfiguration. Static structure specifications with multiplicities $\neq 1$ are considered to be architectural templates for a number of different instantiation possibilities. Before an analysis along our formal model of port-based components is feasible, we need to resolve multiplicity specifications $\neq 1$, and select an appropriate instance with multiplicities 1 only. Of course this is a limitation of our current formal model, but it also highlights potential ways to cope with multiplicity specifications. Instead of an extension of the formal model, multiplicities may also be resolved on the syntactical level, leaving the formal model untouched and thus easier to comprehend.

Under the assumption of fixed upper bounds for port and component multiplicities (instead of unbounded upper limits specified using the UML value *) a static structure with a maximal number of instances is constructed by adding port declarations to components and component declarations to composite components. The resulting specification shows multiplicity 1 only. The names of additional port and component declaration use, for instance, the original name extended by a suffix derived from a counter variable according to the given upper bound. After all declarations have been added, binary assembly connectors are used to link all of the created port declarations. The communication timing «sync» or «async» is inherited from the original connector specification. The approach integrates with sequential adapter connectors, specified by a connector with stereotype «seq-adapter» (Sect. 1), and orthogonal component behaviours, specified by state machines with stereotype «orthogonal» (Sect. 4), by taking into account the new port declaration names within the particular behaviour computation.

2. Component Behaviour and UML State Machines

UML state machines provide a convenient means for the concrete representation of the implementation behaviour of port-based components. We consider a particular, restricted class of UML state machines along features as shown in Fig. 7.4. Their semantics is given by the superstructure specification [**OMG09**]; their informal meaning is discussed also in the reference manual [**RJB05**]. The behaviour of the CoCoME component CashDeskApplication in Fig. 8.9b provides an example which applies all of the mentioned features.

(1) States are initial, simple, submachine, choice or final states.
(2) State entry and exit actions are sequences of send and internal actions.
(3) Transitions comprise guards, triggers and effects.
(4) Guards are boolean expressions component, port or signal attributes.
(5) Transition trigger events are signal events only (signal trigger).
(6) Effects are sequences of send and internal actions.
(7) Submachines follow the same restrictions.

FIGURE 7.4. Relevant state machine features

The only deviation from the UML concerns the "input-enabled" execution semantics that discards events if no matching transition is enabled. There is an event pool associated with a state machine which is used to check for event occurrences. The order of event processing is a semantic variation point, but usually a FIFO order is assumed. Events are ignored, that is, removed from the pool without any effect if the current stable state does not show an outgoing transition whose trigger matches the given event. Stability means that currently there is no run-to-completion step executing [**RJB05**, pp. 604 ff.], [**OMG09**, pp. 565–566]. As illustrated in Fig. 7.5, discarding events corresponds to triggering a self-loop. In particular this means that the machine is not blocked but continues to process received events.

(a) Specification (b) Semantics

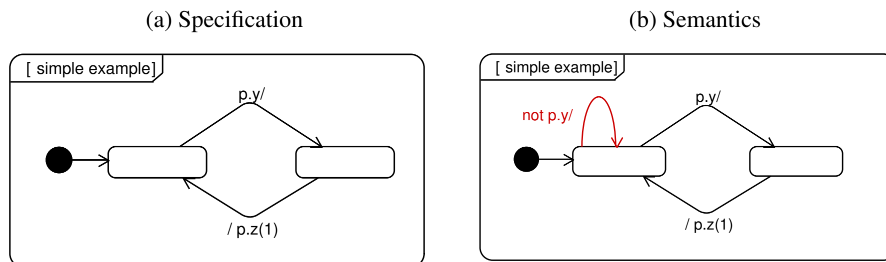


FIGURE 7.5. UML execution semantics for unspecified receptions

Ignoring events conceptually means that received events are definitely lost if a stable state without matching signal trigger is reached. Since such a behaviour is rather an implementation issue, we prefer to use a looser approach that does not rely on a fixed (default) behaviour for unspecified receptions. One of the important aims of the development of our formal component model with asynchronous communication is to prevent unexpected loss of signals by an analysis for communication errors (cf. Chap. 2, Sect. 1.3). For this reason we fix the processing order to use a FIFO policy and interpret a state machine "as-is" without adding any default behaviour for missing event receptions. We rely on our notion of asynchronous output compatibility to guarantee that events that have been sent are not ignored at the receiver's site.

Beyond existing UML features we use additional notations and fix some semantic variation points as follows. Sequences of actions are interpreted as a shorthand for an explicit sequential specification with one transition and successor state for each action. For guards the usual boolean connectivities and simple arithmetic operators such as $+$ or $\leq$ are used. For internal actions we use attribute assignment, simple arithmetic operators and private helper functions. Signal trigger and send actions of state machine transitions use a notation that includes the port name where the particular signal is sent or received:

(1) Signal trigger are written $p.m$/, meaning that the behaviour is ready to receive signal m at port p. The signal's attributes are specified by the operation signature for m as given by the provided interface of port p and may be accessed for data values after this event occurred.

(2) Send actions are written /$p.m(x_1, \ldots, x_n)$, meaning that a signal m is sent via port p. The values represented by variables $x_1, \ldots, x_n$ are assigned to the signal's attributes according to the order of the formal parameters as specified by the operation signature for m in the required interface of port p.

Figure 7.6 shows an example to illustrate the data-related aspects of signal trigger and send actions. A signal trigger of port p:P declared in the context of component type C is defined with regard to the signatures of the provided interface I. The trigger are written p.x/ and p.y/. The values of the latter are accessible using the attributes according to the signal type Y, that is, by an expression y.n. As an example application we consider the send action according to the signature given in the required interface G. The send action is written /p.z(y.n) meaning that we pass the current value of the signal's attribute further with a signal of type Z (not shown in Fig. 7.6) whose attribute value h equals y.n. In the same vein we use constants, simple arithmetics or helper functions to describe attribute values of signals sent.

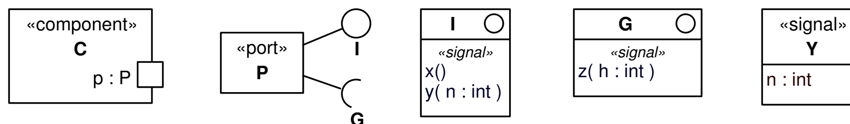


FIGURE 7.6. Data associated with a signal

We define and use two stereotypes, «orthogonal» in the context of a state machine and «tau» in the context of transitions. The latter is a means to specify an anonymous internal action in case further internal details should either be hidden for the purpose of encapsulation or are not relevant for a precise understanding of the behaviour at hand. In contrast to UML internal actions we do not interpret «tau» transition along the usual run-to-completion semantics [**OMG09**, pp. 565 ff.]. If a mixed choice state with both, an outgoing transition with signal trigger and a transition without trigger and only a send action is encountered, the UML semantics always triggers the send transition, even though there is a transition with a signal trigger specified. Figure 7.7 illustrates this point with mixed choice state s0. In order to allow the reception transition to be triggered, we insert a «tau» transition before the send action, in fact resolving the mixed choice state to a state with an internal choice. Note that a state machine interpreter would need to implement the «tau» using a timeout or the like, that decides on the time spent waiting for an event that matches the signal trigger.

The stereotype «orthogonal» is explained along its application in the state machine for the CoCoME component Coordinator in Fig. 8.8. The stereotype is parametrised and is used to derive a common global behaviour of several independent copies of the given state machine that are supposed to be executed in parallel to each other. The parameter

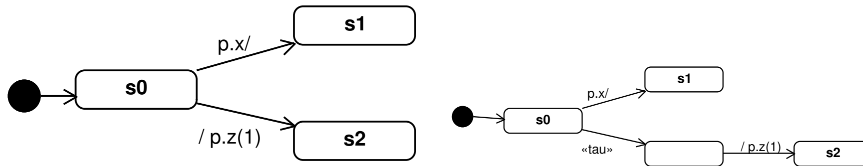


FIGURE 7.7. Internal actions with relaxed run-to-completion semantics

numCopies = |cds| of the stereotype stores the required number of copies. In the concrete example the multiplicity of the set of ports cds as declared by the type specification of Coordinator in Fig. 8.2b is supposed to provide this information. In the translation to transition systems below, the parameter is used to generate indexed and thus different labels.

3. Frame Specifications and UML Protocol State Machines

Figure 7.8 summarise features of UML2 protocol state machines (PSM) that are relevant for frame specifications within our component model. In contrast to state machines, PSMs do not allow for the direct specification of effects. Instead, pre- and post-conditions are supposed to be used to specify behavioural aspects in a more declarative way. Since our formal model does not consider pre- and post-conditions we will, however, not make use of this feature. Therefore there are no action related features in Fig. 7.8. In particular, transitions do not show effects and guards use signal attributes only. Without internal actions we can not specify port or component attribute assignments and therefore it does not make sense to include them in guard expressions.

(1) States are initial, simple, submachine, choice or final states.
(2) Transitions comprise guards and triggers.
(3) Guards are boolean expressions with signal attributes.
(4) Transition trigger events are signal events only (signal trigger).
(5) Submachines follow the same restrictions.

FIGURE 7.8. Relevant protocol state machine features

Making PSMs applicable for the specification of component frames requires extensions in two respects. First, our specifications describe a temporal ordering of messages received and sent. Therefore, we allow transitions with guards, triggers and sequences of send actions analogous to transitions with send actions of UML state machines as described above. Moreover, we allow for the specification of anonymous internal actions using the stereotype «tau» and parameterised specification of multiple copies using the stereotype «orthogonal». Second, we add a further stereotype «opt» for the specification of optional transitions. An *optional transition* must not necessarily exist in the implementation while ordinary transitions will be interpreted as mandatory. It is an obligation for correct implementations to provide a corresponding transition. In contrast to UML state machine semantics, the UML semantics of PSMs considers the reaction to receptions of unexpected events as a semantic variation point [**OMG09**, pp. 538]. Thus, our direct interpretation for the translation of PSMs to PIOs described below is in accordance with the UML semantics.

As an example consider the frame specification of a bank component, depicted in Fig. 7.9. The example is taken from the CoCoME in Chap. 8 and included here for convenience. The specification describes a simple protocol for PIN validation with a succeeding debit of the particular account. The positive answers to the respective requests are both optional. By this means, implementations that always deny a validation or debit request (or both) are correct with respect to the given frame specification. In contrast, the transitions

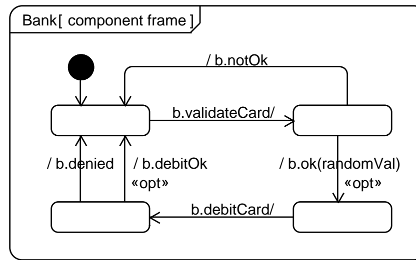


FIGURE 7.9. Frame specification of a bank component

with trigger for the signals validate and debit as well as transitions with negative answers are mandatory for correct implementations.

4. From State Machines to Transition Systems

Even within our restricted class of simple UML (protocol) state machines there are some features which can not immediately be translated to I/O-transition systems (IOTS), input-persistent I/O transition systems (PIO) respectively. State machines use guards and attributes which both are not available with transition systems. Moreover, state machines use a form of hierarchical construction with submachines that neither exist within our formalism.

Guards and attributes need to be manually translated in a preprocessing step either by abstraction or by instantiation in case of finite data domains as given, for instance, by boolean typed variables or enumerations. The translation of submachine states is, due to their macro semantics straightforward by complete expansion and could be automated. In the following we describe the translation from state machines (STM) to IOTSs and after that extend the approach for the translation of protocol state machines (PSM) to PIOs. The necessary preprocessing comprises the following steps:

1. Abstract data variables
2. Abstract and resolve guards
3. Expand submachine states

We explain these steps by pattern-like translations. First, we consider labels involving data variables such as a send action of the form /p.z(x). If the type of x has a finite domain one may generate labels and transitions for each value within this domain. If no guards are involved the distinction does not play a role and it suffices to consider just one abstracted transition labelled /p.z without any parameter value. This pattern is exemplified in Fig. 7.10a. In case succeeding transitions depend on the values sent, for instance using guards that involve the variable x, the concrete values need to be taken into account for each of these transitions. Figure 7.10b shows the corresponding situation after a signal has been sent. The translation generates internal labels and transitions that allow to distinguish the different paths. This approach, however, is not correct if the receiver behaviour also depends on the attributes of the signal sent. For this case, exemplified by a receiver as given in Fig. 7.10c, the transitions need to be aligned such that both, sender and receiver, use matching transition labels. The sender in Fig. 7.10b then uses transitions labelled /p.zPos and /p.zNeg0 instead of the given send transition with succeeding internal transitions.

The translation of UML submachine states is straightforward by expanding the particular state as illustrated in Fig. 7.11. The transition p.cancel/ is added to any stable submachine state. The submachine transition to its final state triggers the unlabelled completion transition to state s1.

The result of the preprocessing is an intermediate STM with new labels, transitions and states, that maps directly to an IOTS. The states of the IOTS, including the initial state are given by the STM states. STM signal trigger translate to input labels, the send actions to

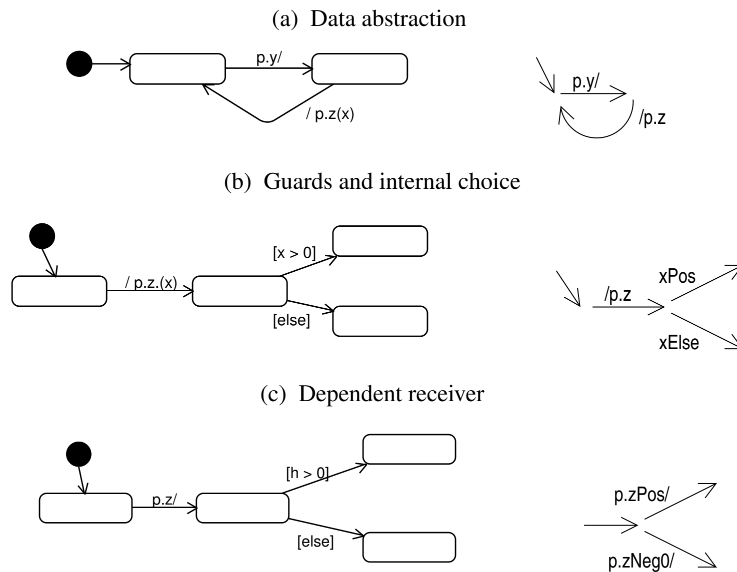


FIGURE 7.10. Patterns for the translation of data variables and guards

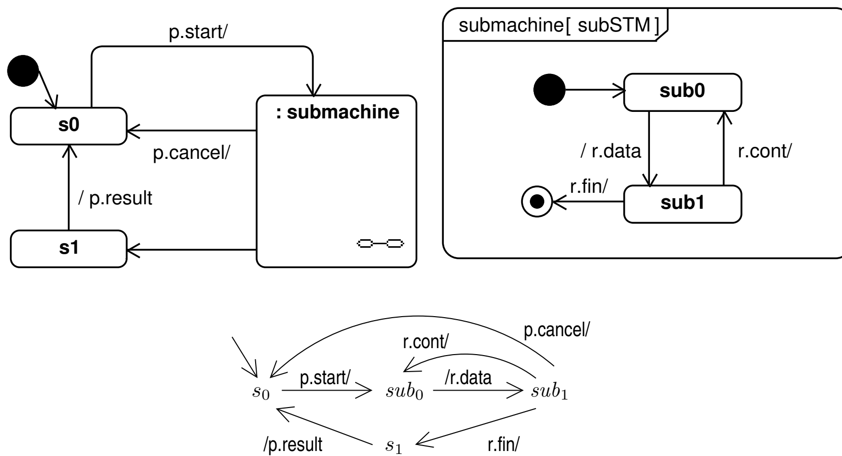


FIGURE 7.11. Expansion of submachine states

output labels and the internal actions of the STM to internal actions of the IOTS; transitions are in 1:1 correspondence including «tau» transitions of the STM which are translated to τ-transitions of the IOTS. The result is an IOTS with an I/O-labelling that comprises only labels that have been used in the state machine specifications. Moreover, to assign correctly to a partitioned I/O-labelling requires to use the port-prefixes in the STM labels. Alternatively, and more complete, one could use the static structure specification and create a partitioned I/O-labelling using the information available with the port declarations of the context classifier of this STM.

The translation is complete, if the original STM was not specified to be «orthogonal»; otherwise a product is computed along the required number of copies given by the stereotype's parameter numCopies. More formally, let A be the IOTS obtained by translation of the STM. Let $(\rho_i)_{i \in I}$ be a family of relabellings (with index set I) such that each ρ_i yields globally unique new labels by appending i to the original labels. Then the translation with

respect to a stereotype «orthogonal» with parameter numCopies is given by the product $\otimes_{i \in \{1,\ldots,|\mathsf{numCopies}|\}} A\rho_i$. Consider for example the STM of the Coordinator component in Fig. 8.8. The port declaration cds:C-CD in Fig. 8.2b uses multiplicity *. Assume this multiplicity is fixed to some value $k \in \mathbb{N}$, then the product consists of completely interleaved transitions that have been relabelled using $(\rho_i)_{i \in \{1,\ldots,k\}}$. The relabelling generates labels of the form l_i for all $l \in \bigcup L_A$ for all $i \in \{1, \ldots, k\}$.

Since protocol state machines are special kinds of STMs, the translation of PSMs to PIOs works identically, except the treatment of the stereotype «opt» which has not be considered in the context of STMs. However, the necessary extension is minimal. Let $A = (L, S, s_o, \Delta, \Pi)$ be a PIO. Translate all state machine transitions to transitions in Δ and if the annotation «opt» is absent include the given transitions in Π.

Based on the given translation we can use UML state machines and protocol state machines for the concrete specification of component behaviour and frames with a precise meaning on the formal level of IOTSs and PIOs. The connection between concrete UML specifications and their formal meaning can be given by an instantiation of the abstract model of frames and behaviours developed in Chap. 6. We assign the translation of PSMs to provide the frame and the translation of STMs to provide the behaviour of a component. Denote the translation of STMs to IOTSs by $[\![\cdot]\!]^{\mathrm{IOTS}}$; and of PSMs to PIOs by $[\![\cdot]\!]^{\mathrm{PIO}}$.

Definition 7.1 (UML frame and behaviour) *Let* Frm *be a protocol state machine for component $C \in \mathrm{Cmp}$. Let* Impl *be a state machine for simple component $SC \in \mathrm{SCmp}$. The* frame *of C is given by $frm(C) = [\![\mathsf{Frm}]\!]^{\mathrm{PIO}}$. The* behaviour *of SC is given by $beh(SC) = [\![\mathsf{Impl}]\!]^{\mathrm{IOTS}}$.*

The translation can be used to develop a UML theory of compatibility and refinement for the considered classes of STMs and PSMs. For instance, a concrete notion of UML protocol conformance [**OMG09**, pp. 534–535] could be based on the theory for PIOs and IOTSs developed in Chap. 4. In this way, also notions of composition and compatibility could be defined, both, for STMs and PSMs which paves the way to extend existing UML modelling tools to provide convenient support for the UML-based design and verification of port-based component systems.

We summarise the translations described within this chapter by an abstract algorithm that is used for both, the translation of STMs and the translation of PSMs. Let M be a STM or PSM. The translation $[\![\cdot]\!]^{\mathrm{PIO}} : \mathbf{STM} \cup \mathbf{PSM} \rightarrow \mathbf{PIO}$ is given by $[\![\mathsf{M}]\!]^{\mathrm{PIO}} = A$, where the construction of A is described by the following algorithm:

```
M ← preprocess(M)
let A = ((I_1, O_1, ..., I_n, O_n), T), S, s_0, Δ, Π)

(I_1, O_1, ..., I_n, O_n) ← port declarations of static structure
T ← internal labels of M
S ← states of M
s_0 ← initial state of M
forall transitions (s, l, s') in M do
     If (l = ε ∧ «tau») then Δ ← Δ ∪ (s, τ, s')
     else Δ ← Δ ∪ (s, l, s')
     If (¬ «opt») then Π ← Π ∪ (s, l, s')
od

If («orthogonal») then
    let n = numCopies
    let (ρ_i)_{i∈{1,...,n}} be a family of i-appending relabellings
    A ← Aρ_1 ⊗ ... ⊗ Aρ_n
fi
```

We encoded the absence of a label in UML transitions annotated «tau» by $l = \varepsilon$. Moreover, the relabellings may not use an arbitrary strategy for the creation of unique labels, as the product must later be relabelled to synchronise with a corresponding communication partner. But then one needs to know about the concrete labels, i.e., about the applied strategy. Finally, note that $[\![\mathsf{M}]\!]^{\mathrm{IOTS}}$ is now given by $[\![\mathsf{M}]\!]^{\mathrm{PIO}}$.

CHAPTER 8

The Common Component Modelling Example (CoCoME)

1.	**Modelling of the CoCoME**	**125**
2.	**Static Structures**	**126**
3.	**Component Behaviours and their Translation**	**129**
4.	**Hierarchical Component Behaviours**	**131**
5.	**Frame Specifications of Simple and Composite Components**	**136**
6.	**Analysis and Proof Obligations**	**137**

The Common Component Modelling Example (CoCoME), initiated as a GI-Dagstuhl research seminar[1], was a modelling contest that aimed at a comparison of existing component models using a common requirement specification and reference implementation of a point-of-sale system. Within this chapter we revise our initial submission and illustrate features of our component model that were not available at the time of writing [**KJH$^+$08a**]. The formal model sketched in [**KJH$^+$08a**] considered only synchronous communication based on a rendezvous mechanism. It did not feature a theoretical model for refinement and compatibility and it did not distinguish between behavioural specification (frames) and implementation (behaviours) of components. After a brief introduction into the CoCoME we discuss static structure specifications and illustrate the description of component behaviours with UML state machines and their translation to I/O-transition systems. The modelling of more complex component behaviours is illustrated using UML submachines. Finally, we introduce frame specifications for simple and composite components using UML protocol state machines and discuss the analysis of compatibility, implementation correctness and refinement on the level of the corresponding transition systems.

1. Modelling of the CoCoME

The original requirements and modelling of the trading system underlying the *Common Component Modelling Example* (CoCoME) is due to Larman who applied the system to illustrate an approach to object-oriented analysis and design in [**Lar04**]. In [**HKW$^+$07**], the example is used to provide requirement specifications for the participants of the modelling contest CoCoME. The trading system models sale and management of products within stores that may belong to a common enterprise. The system comprises an embedded system part with cash desks and associated hardware devices such as a cashbox and a card reader, as well as an information system part for product management within an inventory. Also connections with external systems are taken into account by a connection to a bank server to support card payment at a cash desk.[2]

The following summarises the key elements of the system as described in [**HKW$^+$07**, pp. 16–18]. A *cash desk* comprises the following hardware devices: bar code scanner, card reader, cash box, printer, light display, and a cash desk PC to which the single devices are connected. The PC also manages connections to an external bank server. A *cash desk line* denotes the set of cash desks that belong to a particular *store*. A server for the

[1] `http://www.cocome.org`

[2] Note that such a requirement calls for the modelling of an open system.

product management along an *inventory* is associated with the store. Several stores might be organised within an *enterprise* with which again a server is associated. The server allows to manage the product stocks of different stores. For the behavioural modelling in the succeeding sections we focus on the functional requirements according to the use cases *process sale* (UC1) and *manage express checkout* (UC2) [**HKW+07**, pp. 19–21]. The former describes the standard process and exceptional processes for selling products at a cash desk, and the latter describes how a so-called cash desk express mode should be enabled or disabled automatically, depending amongst others on the number of items sold at a certain cash desk within a certain amount of time.

Two important features of our component model are its strict use of ports to specify the potential interaction points of components and the distinction between types and declarations. These features already enabled us to model some aspects of the trading system in a more convenient way, compared to the static modelling as given by [**HKW+07**]. Besides minor deviations from the given behavioural requirements, the main deviation from the CoCoME requirements is a structurally different modelling in the information system part. Since we will not discuss this part of the CoCoME system, we refer to [**KJH+08a**] for further details. In the following we focus on the embedded system part of the CoCoME and aim at illustrating features of our component model that were not available at the time of writing [**KJH+08a**].

Modelling Approach. Due to the modelling of component behaviours using UML state machines, which we consider to be representations of the *implementation* of a component, our original approach to the design and implementation of the CoCoME essentially proceeded bottom-up. We started from the given use case descriptions and sequence diagrams as a representation of detailed requirement specifications. After and during static structure modelling for simple (non-composite) components we designed component behaviours accompanied by formal analysis of the respective assemblies. The simple components were applied for the design of composite components, yielding a first draft of the complete architecture. Within the next iterations, the alternative and exceptional processes of the use case descriptions were taken into account to extend and correct the initial design. In case of ambiguous or unclear requirements our design followed the prototype implementation of the CoCoME.

Since we didn't develop frame specifications, there was no top-down step (in the sense of Chap. 6 and Prop. 6.9) involved during system development. In order to argue for a bottom-up step in the sense of Prop. 6.8, the initial requirement specifications could be considered to represent an assembly frame specification for the CoCoME system. Then, even though implementation correctness was not discussed formally, the development followed a bottom-up approach, because all the developed component behaviours were informally discussed at length in [**KJH+08a**], [**KJH+08b**] respectively, with respect to the behavioural requirements of the CoCoME. It is characteristic for bottom-up design to develop single components in great detail and afterwards compose these components to obtain a communicating system.

2. Static Structures

In order to provide a clear-cut entry point for the succeeding modelling we describe the static specification of the component Store which is one of the composite components in the upper layers of the hierarchical system design. As depicted in Fig. 8.1 the component Store is a composition of a CashDeskLine, an Inventory and an instance of the component Data. The inventory is connected to a facade component Data which hides the concrete data base from the application as required in the CoCoME. The port e:Enterprise of Data is not used, when instantiating the component as part of a Store. Also, the optional operator ports of Inventory, allowing to attach explicit GUI components, remain unconnected. The

component Store uses mandatory relay ports to connect to a bank component and a data base. Relaying the port I-PD of component Inventory is optional, in order to take into account the requirements of the exceptional processes in Use Case 8 (enterprise server may be temporally not available). The bank component, required for card payment at the cash desks of a store, is considered external to the system. Therefore the component declares a relay port of type CDA-Bank to delegate incoming and outgoing communications between a bank and internal components, respectively. The port multiplicity with a lower bound of 1 indicates that for a proper instantiation of Store it is strictly required to provide appropriate bank connections.

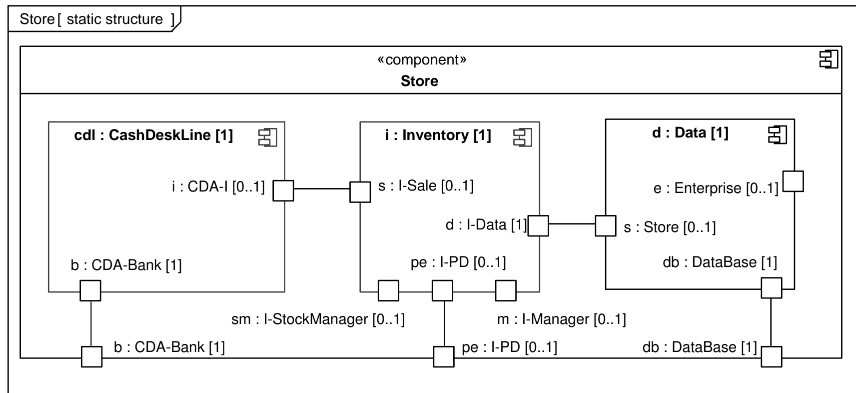


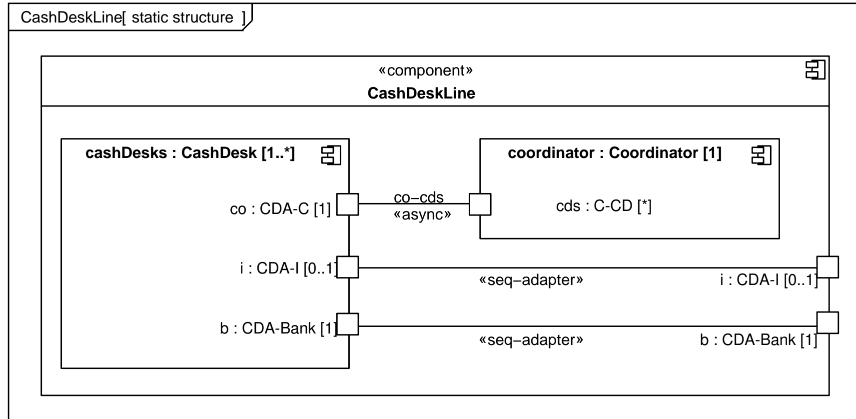
FIGURE 8.1. Static structure of the component Store

Any store instantiated as part of the trading system comprises a cash desk line which in turn represents a set of cash desks, monitored by a coordinator. A CashDeskLine (Fig. 8.2a) consists of at least one cash desk connected to a coordinator which decides on the express mode status of the cash desks. The composite component CashDeskLine declares two relay ports delegating the communication between the cash desks, the inventory (i : CDA-I) and the bank (b : CDA-Bank). The connector declarations in Fig. 8.2a are annotated with the stereotype seq-adapter, meaning that the communication between the ports of the cash desks and the relay ports i and b respectively, is implemented by a sequential adapter. In contrast, the communication between the cash desks and the coordinator does not need to be adapted, because each CashDesk instance is linked via its CDA-C port to its own instance of the coordinator port C-CD. Sharing the bank connection among the desks of a cash desk line follows the CoCoME requirement in [**HKW+07**, Fig. 5] which shows a multiplicity of 1 for the particular required Bank interface, port respectively.

The coordinator component, specified in Fig. 8.2b, decides on the cash desks' sales mode (express or normal mode). The component declares its port of type C-CD with multiplicity * to allow to connect an arbitrary number of cash desks that should be monitored by this coordinator. An asynchronous connector «async» is used to link the components. By this means the concurrently operating cash desks communicate along FIFO buffers with the coordinator component.

The CashDesk component as depicted in Fig. 8.3 is the most complex composite component of the trading system. Each cash desk consists of several hardware devices managed by a cash desk PC. The cash desk specification models the embedded system part of the CoCoME with characteristic features of reactive systems such as asynchronous message exchange or distinguished controller components at the centre of acyclic architectures. The cash desk component comprises six subcomponents modelling the hardware devices as described in the CoCoME and one component CashDeskApplication (CDA) modelling the functionality associated with the cash desk PC. We consider rendezvous-based as well as

(a) CashDeskLine



(b) Coordinator

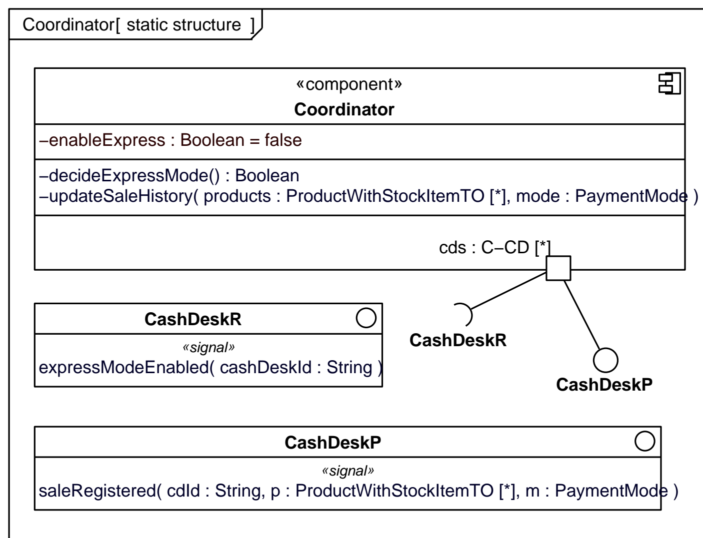


FIGURE 8.2. Static structure specifications

FIFO buffered message exchange and therefore the communication timing of the assembly connectors is left unspecified. For the behavioural description and analysis we will consider both cases, «sync» and «async».

A cash desk has three ports to allow for the communication with a bank, inventory and coordinator component. The component and port multiplicities of the static structure in Fig. 8.3 reflect the requirements of the CoCoME. Since an exceptional process for Use Case 1 (process sale) explicitly mentions that the inventory might not be available, the port i is specified optional. The optional ports of CashBox, Scanner and CardReader model the communication of an operator with the particular hardware device. In case of a cash desk actually deployed, the ports might be connected with some low-level interrupt handler. The ports are not part of the external interface of a cash desk component [**HKW+07**, Fig. 13] and therefore do not delegate to corresponding open ports of CashDesk. For a translation to our formal model either further components could be connected or unary connectors would be applied to close these ports.

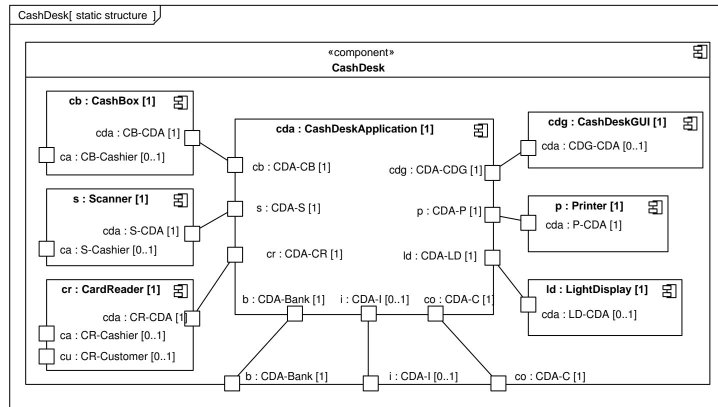


FIGURE 8.3. Static structure specification of CashDesk

The cash desk application links all internal components of a cash desk and communicates with components external to the cash desk such as a bank server, the inventory or the cash desk coordinator. In order to facilitate the particular communication, CashDeskApplication declares one port for each. Figure 8.4 (top) shows the static structure of the component with private state attributes, port declarations and interface details. As a naming convention, we have used component abbreviations such as CDA-CB for the port types and the suffixes R (P) for interfaces required (provided) by a port. Component attributes are used as helper variables within UML state machines for a more convenient specification of component behaviour. In the same vein we also use port attributes which, however are not shown in the diagram of Fig. 8.4. A fully detailed static specification can be found in [**KJH+08b**].

The static structure specifications of the remaining subcomponents of the composite component CashDeskLine essentially mirror the ports and interfaces of the central component CashDeskApplication and are not shown here. Again, fully detailed static structures are given in [**KJH+08b**]. Note, however, that the specifications in [**KJH+08b**] might slightly deviate from the specifications given within this chapter in order to align the modelling with new features such as asynchronous FIFO buffered communication.

3. Component Behaviours and their Translation

We consider the implementation of the composite component CashDeskLine, CashDesk respectively and start with basic state machines for the simple components of CashDesk to illustrate the translation of basic features, proceeding step-wise to more complex behaviours due to the application of more features. We omit a presentation of the component CashDeskGUI since, according to the reference implementation [**HKW+07**], the GUI allows its clients to send messages and signals in any order, that is, without any behavioural restriction. The rather complex behaviours of the components CashBox and in particular CashDeskApplication are discussed without translation in Sect. 4.

The most basic component within the CashDesk component is modelled by the bar code scanner whose implementation behaves as depicted in Fig. 8.5. At the cashiers port ca the scanner component receives the barcode of a scanned product which is then sent further along port cda to the cash desk application. The signal trigger is given by an event scanProductBarcode which is according to our conventions for the derivation of signal types (cf. Chap. 7) an object of type ScanProductBarcode with attribute barcode:int. The translation to an IOTS is immediate after an abstraction of the data represented by the signal attribute barcode:int that is passed to the cda port.

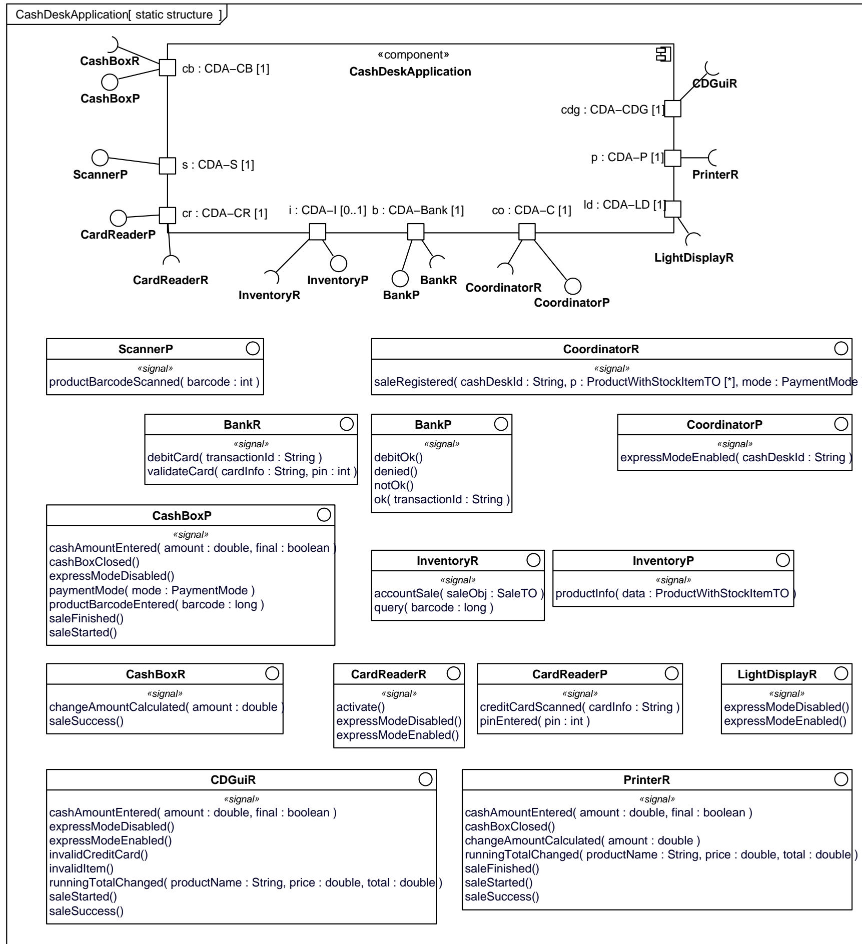


FIGURE 8.4. Static structure specification of CashDeskApplication

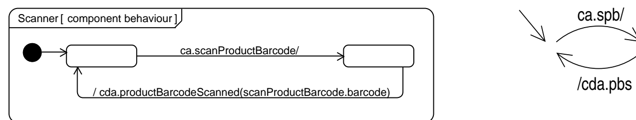


FIGURE 8.5. Behaviour of the Scanner component

Next we consider the behaviour specification of of the component LightDisplay. Cash desks may be switched to an express mode which is signalled at the cash desks LightDisplay. Both signals, expressModeEnabled and expressModeDisabled, are always accepted. Initially only the former triggers a state change, encoding the switch to express mode and hereafter only the latter triggers the state change to normal mode back again. The component behaviour specifies internal behaviour using private operations that represent signals for the hardware device to show green and black lights. The private operations are interpreted by internal labels green and black, and then there is again a direct mapping to an IOTS.

The behaviour of the Printer component is specified in Fig. 8.7 and illustrates a specification with guards. The printer records after a sale started the last registered product by its name and price and prints the current total price. After the sale is finished, the behaviour

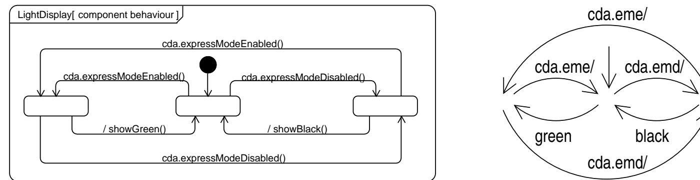


FIGURE 8.6. Behaviour of the LightDisplay component

waits to receive information concerning the amount of money paid by the current customer. As long as this information is not explicitely finished, a loop is entered that is left either if a saleSuccess event is received or the the entered amount is explicitely finalised. After that change is calculated and the the closing the of cash box registered. The choice state uses a boolean guard that is resolved to a loop in the IOTS in a straightforward way.

Figure 8.7b shows the behaviour specification of the component CardReader as a first example of a more complex behaviour. There are two main functional regions in the behaviour specification. On the one hand, the card reader may be deactivated by receiving a message expressModeEnabled. On the other hand, within the lower region reached by the reception of a signal activate, the port possibly engages in sending credit card information. From any of the "active" states the card reader may be again be deactivated by the message expressModeEnabled. The IOTS-translation is depicted in Fig. 8.7d. The transitions for reception of expressModeEnabled and activate leaving the complex state in UML state machine are translated to one transition for each of the substates in the IOTS-translation.

The behaviour specification of the component CashBox is more involved compared to the other devices attached to the CDA, but it does not introduce new features. Therefore, we defer the discussion of its behaviour to Sect. 4. The same holds for the specification of the component CashDeskApplication. Even though this component plays an application-specific key role as the centre of a star topology (within CashDesk), we directly switch to the Coordinator component which resides on the hierarchical level of CashDesk.

The CashDeskLine of a store consists of a number of cash desks and an instance of the component Coordinator whose behaviour is specified in Fig. 8.8. The coordinator decides on the mode of the cash desks which is either express or normal. Note that even if the coordinator decided to signal expressModeEnabled, the component may receive yet another sale registration from the same cash desk because the communication partners are executing concurrently. In this case the sale registration has precedence over the coordinator's decision: the behaviour receives the signal saleRegistered and recomputes its internal decision. The component keeps track of the particular sale history for each cash desk and decides upon this history to signal an express mode switch for this particular cash desk.

The update of the sale history is required to be synchronised (not described in the behaviour) due to the potentially concurrent execution of several requests via port declaration cd:cds (specified with multiplicity *). Due to this feature, the STM specification uses the stereotype «orthogonal» and assigns the required number of copies of the given behaviour to the multiplicity of cd:cds. The translation takes this parameter into account and computes a corresponding product of the translated transition system. The computation applies a relabelling of each copy such that the final transition system results in a complete interleaving of the |cds| copies.

4. Hierarchical Component Behaviours

The state machines for the components CashBox and CashDeskApplication result in transition systems too large to be depicted graphically. However, we believe that the behaviours nicely illustrate a real-world application of our component model and therefore we include a brief presentation, though without a translation to transition systems. The

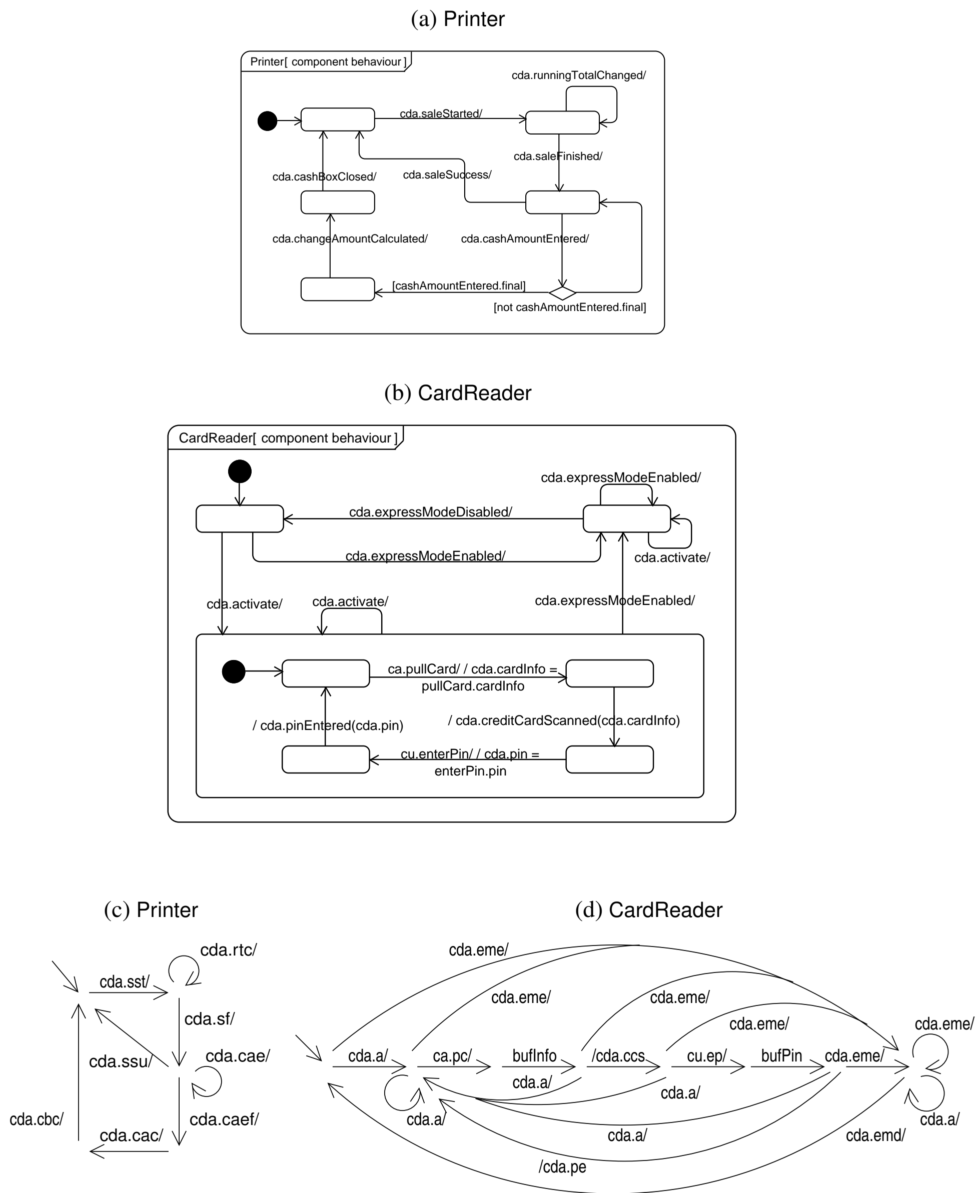


FIGURE 8.7. Behaviour of the Printer and the CardReader component

examples clearly indicate the importance of hierarchical constructions and guards for a comprehensible modelling of more complex behaviours.

The component CashBox uses two ports, an optional operator port ca of type CB-Cashier and a mandatory port cda of type CB-CDA; cf. Fig. 8.3. The former allows to connect the component to inputs of the cashier, the latter specifies the behaviour with regard to the cash desk application. The message productBarcodeEntered at the CB-Cashier port is not mentioned in the sequence diagrams of the CoCoME but in the standard process and also in an exceptional process of Use Case 1. It allows the cashier to manually enter a product bar code in case there are problems with the bar code scanner. The component behaviour of CashBox in Fig. 8.9a maps the cashier's input at the ca port essentially directly to corresponding notifications at the cda port.

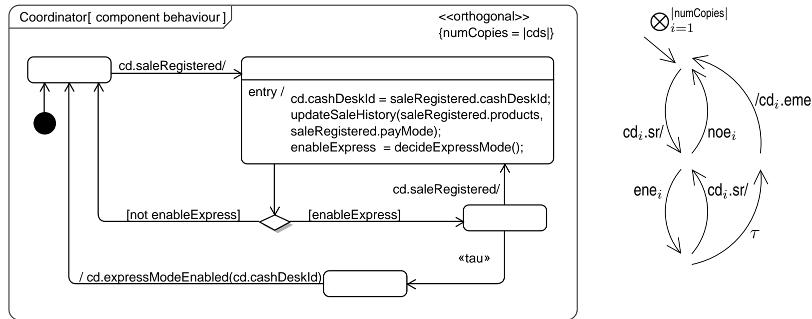
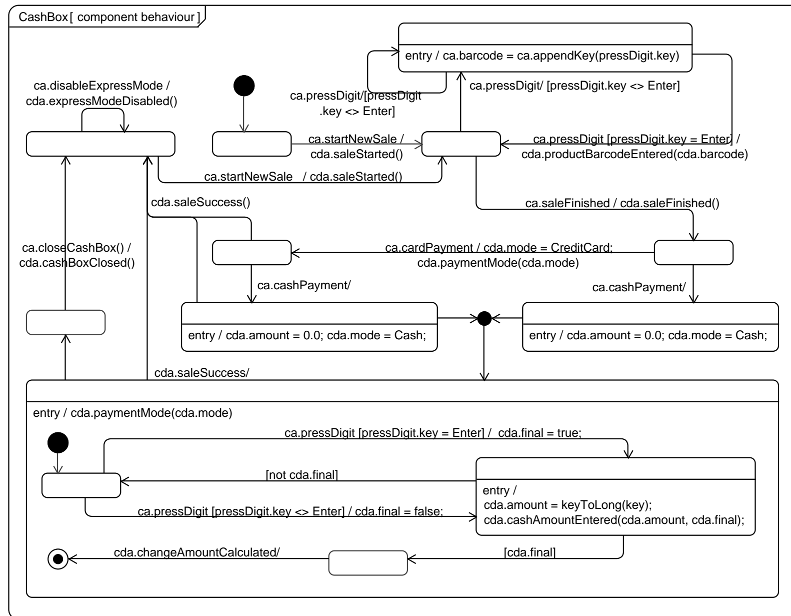


FIGURE 8.8. Behaviour specification of Coordinator

Figure 8.9b specifies the component behaviour of the cash desk application. Using the port declarations of the static structure in Fig. 8.4 it shows the dependencies and interlinkages between the different ports of the CDA. For example messages sent via ports p or cdg such as p.saleStarted and cdg.saleStarted are sequentially arranged after the message cb.saleStarted was received at port cb. Furthermore port attributes as well as component attributes such as itemCounter are assigned as an effect, and afterwards used, for instance, as actual parameters in messages sent.

Since the specification of the cash desk application's behaviour is rather involved, we used submachines, shown in Fig. 8.10, to factor out the major steps of the entire sale process: after saleStarted was received at the cash box port cb, the submachine ScanItems repeatedly receives product bar codes and notifies the printer and the GUI about product details such as name and price. Thereafter the payment mode must be chosen, resulting in a transition to the corresponding submachine. Note that the execution of CardPayment might be cancelled at any state by the reception of cb.paymentMode(Cash) modelling the cashier's ability to switch from card to cash payment, e.g., in case of problems with the credit card. Then, before the sale process is completed, the component tries to account the sale at the inventory within AccountSale. If it is not available (which is an explicit requirement of the CoCoME), the sale information is stored locally and delivered during the next sale processes. Finally, in SwitchMode the component waits for a signal to switch into express mode, to disable a previous express mode, or to start a new sale.

(a) CashBox



(b) CashDeskApplication (cf. Fig. 8.10 for submachines)

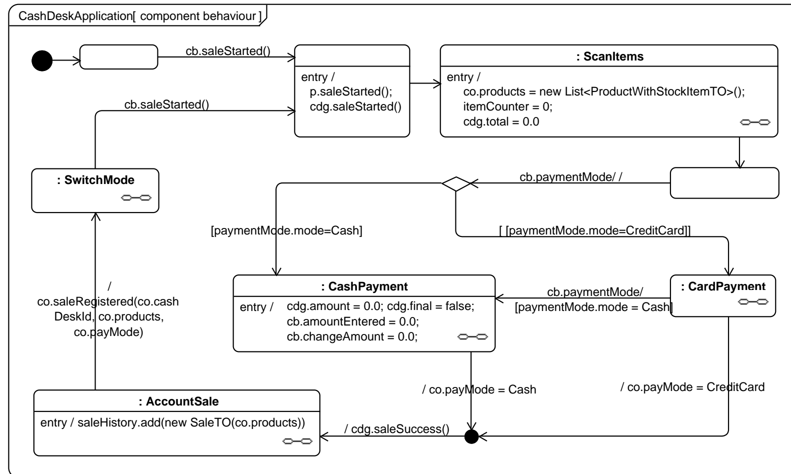


FIGURE 8.9. Behaviour specifications of CashBox and CashDeskApplication

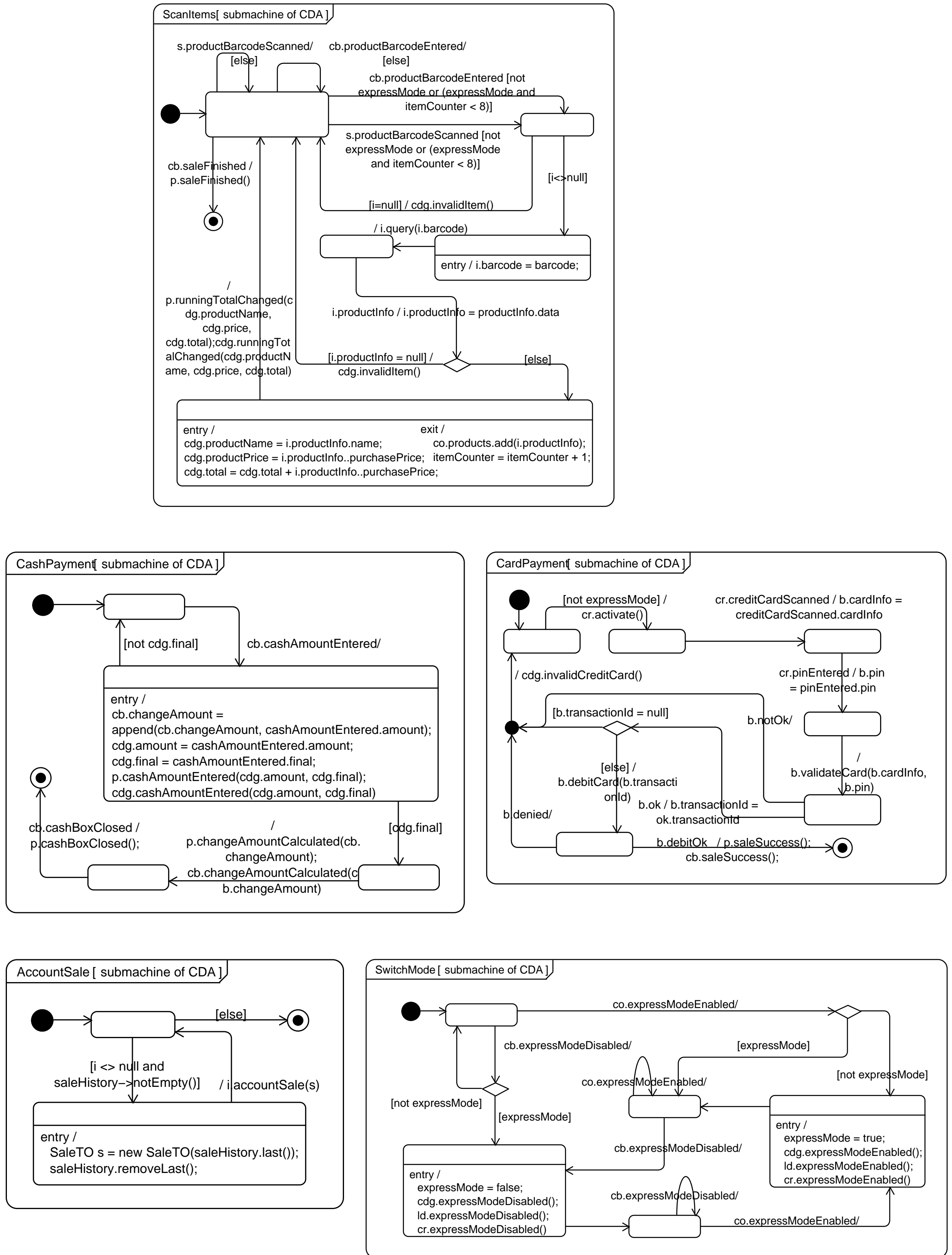


FIGURE 8.10. Submachine specifications for CDA (cf. Fig. 8.9b)

5. Frame Specifications of Simple and Composite Components

Frame specification provide means for more abstract specifications of component behaviour. Internal actions are not distinguished, abstracting from internal local computations and additional annotations may be used that specify optional transitions. We use a slightly extended form of UML protocol state machines for the concrete specification of component frames. The applied features and extensions are explained in Chap. 7 together with a translation to input-persistent I/O-transition systems. In the following, we discuss frame specifications for the simple components Coordinator and Bank, and for the composite components CashDesk and CashDeskLine.

(a) Coordinator (b) PIO translation

(c) Bank

FIGURE 8.11. Example for simple frame specifications

The specification of frames takes requirements[3] into account without focusing on implementation details. In this way, a more abstract design model with respect to the given requirements is achieved. Additionally, parts of the behaviour may be specified optional to allow a wider range of correct implementations. The specification for the component Coordinator in Fig. 8.11 illustrates these aspects. The design was driven by the requirement to continuously be able to receive a signal saleRegistered. The component internally decides either to send a signal expressModeEnabled or to receive yet another sale registration. In contrast to the optional decision to enable a cashdesk's express mode, the internal transition moving back to the initial state is mandatory. Otherwise implementations would be allowed that stop after having received the first signal saleRegistered.

Compared to the behaviour of the coordinator component in Fig. 8.8, the frame specification is less detailed and thus better comprehensible. As the frame of a component is supposed to include all of the potential communication behaviour of an implementation, it is perfectly applicable for the construction of assembly frames that are smaller and again better comprehensible compared to assemblies of component behaviours.

[3] given, for instance, by use case descriptions or sequence diagrams (or both)

The figure also illustrates the PIO translation of the protocol state machines according to Chap. 7. Beside the mapping of transitions without stereotype «opt» to persistent PIO transitions, the translation is analogous to the IOTS translation of UML state machines that was illustrated in Sect. 3 above.

With Fig. 8.11c we discuss a different scenario for the modelling of a frame specification. The CoCoME is an example of an open system due to the communication with a bank component that is external to the trading system. A concrete component Bank is not available. Nevertheless, the communication for credit card payments is modelled as part of the behaviour of the component CashDeskApplication. In this case, a frame specification for the bank component can be used to specify the expected behaviour of the yet unknown component. Such a specification states that any component, implementing the given frame correctly is allowed to be integrated with the trading system.

The concrete specification in Fig. 8.11c allows implementations that always refuse credit cards or always deny the debit of a successfully validated credit card. This optionality is motivated by the possibility to switch to cash payment at the cash desk of a store. Thus, a sale process may be finished even in case a bank component always refuses the validation of a credit card.

As an example for frame specifications of composite components consider the static structure specification of the component CashDeskLine in Fig. 8.2a. The component declares two ports, one for the communication with an inventory and another one for the communication with a bank. With frame specifications of composite components we may abstract from any internal details, i.e., we specify the behaviour as described by the given requirements, but only with respect to the communication at the given ports. For instance, Fig. 8.12a shows a component frame that takes the optionality of credit card payment into account. In order to simplify the presentation we did not model the optionality of the communication with the inventory as required by the CoCoME. The frame specification shows a completely optional internal decision to sale products in normal or in express mode. In normal mode, credit card payment is possible and the bank communication is integrated by the submachine state BankComm.

The frame specification of CashDesk in Fig. 8.12b demonstrates a decomposition of the frame specification of CashDeskLine and hence is an example for a top-down design step. The only difference is an additional part for the communication with the coordinator component. Intuitively, the composition of the frames of the subcomponents CashDesk and Coordinator is supposed to result in a transition system with the additional part being completely internal. In this case, the assembly behaviour would be a valid blackbox refinement of the frame of CashDeskLine and the decomposition a valid top-down step in the hierarchical design of the trading system.

6. Analysis and Proof Obligations

The formal analysis of the given component behaviours and frames can be split into three parts. First, behaviour analysis, considering component behaviours and their composition within assembly behaviours only. Second, correctness analysis for simple and composite components, and third, frame analysis that checks for refinement and properties such as communication-safety using component and assembly frames.

Behaviour Analysis. Our original submission of the CoCoME focused on correctness analysis between port protocols and component behaviours, and neutrality checks within the component CashDesk. We do no repeat this analysis here. Instead, we briefly mention specifics that relate to the component model as developed within this thesis.

First of all, the approach to reduce the internal structure of CashDesk by neutral component behaviours still works using Chap. 3. However, in case we are interested to show

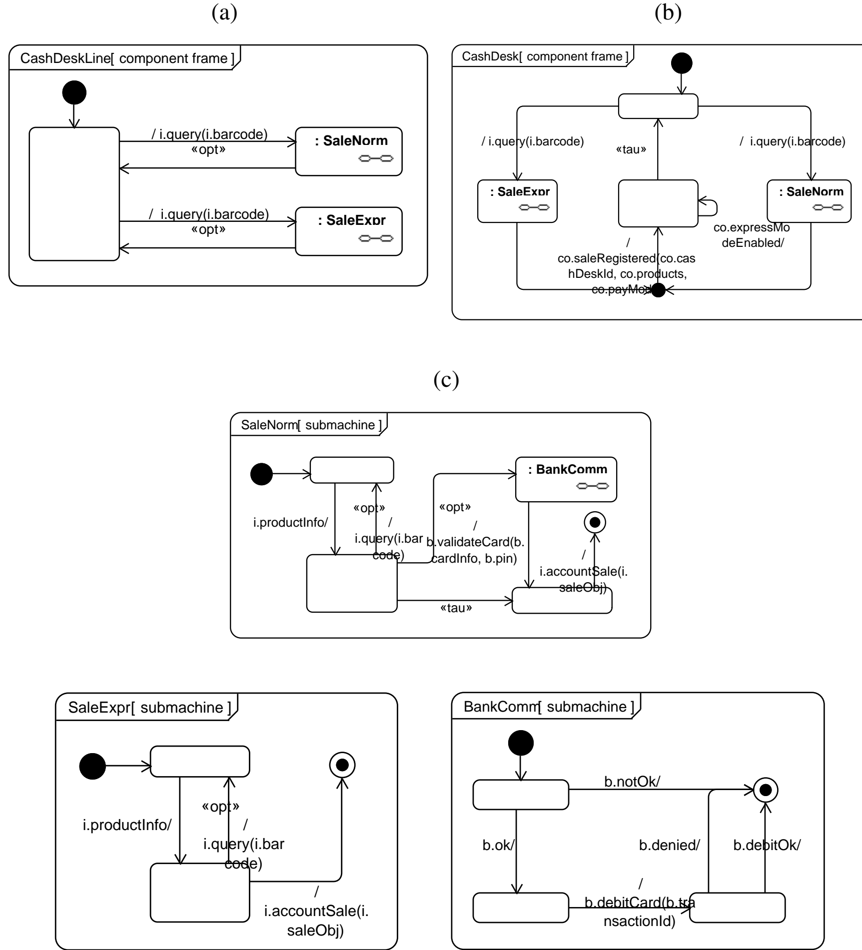


FIGURE 8.12. Frame specifications of composite components

communication-safety of the given assembly, we may not use the syntactically reduced assembly. Our approach disregards the communication direction in the checks for neutrality. Therefore, if the central component CashDeskApplication is not compatible to the output of an attached device, for instance the cash box component, then this incompatibility disappears in a reduced assembly as long as the behaviour of the attached component is neutral for the CashDeskApplication.

In order to verify communication-safety, it is necessary to consider the complete assembly $asm(\mathsf{CashDesk})$. For the case of weak comm-safety, pairwise verification using Prop. 4.37 suffices. Ultra-weak comm-safety, however, must be checked in an incremental way using Prop. 4.40 that computes the global assembly behaviour stepwise.

While the initial modelling in [**KJH**$^+$**08a**] used synchronous connectors, we use asynchronous connectors now. Such a modelling is closer to the CoCoME requirement of using an event bus for the communication between the hardware devices of a cash desk component. Hence, the underlying behaviours are completely buffered and, for instance, an analysis of weak comm-safety with Prop. 4.37 results in an obligation to proof the following conjunction:

$$\bigwedge_{c:C \in leaves(asm(\mathsf{CashDesk}))} \Omega(beh(\mathsf{cda{:}CashDeskApplication})\xi)\alpha \leftrightarrow_w \Omega(beh(c:C)\xi)\alpha\ .$$

In general, it is necessary to check all connected pairs of components. Thus, due to the specific topology of the given component, we can use the *leaves* of the assembly to obtain all relevant pairs.

The example is a suitable candidate to evaluate the CFSM-based approach with a symbolic verification of asynchronous compatibility; cf. Chap. 5. However, due to the size of the underlying transition systems, a manual check is not feasible and must be postponed until mechanical tool support is available.

Correct Implementation. Correctness analysis involves refinement proofs between the behaviour and the frame of a component. For the case of simple components, we need to check blackbox refinement between the PIO-translations of the given UML specifications. For instance, the behaviour of component Coordinator as given in Fig. 8.8 is a correct implementation of the frame specification as given in Fig. 8.11a. Consider the translation to transition systems and ignore for now the product operator and the parameter i in the labels. Assume, that the states of the behaviour transition system are labelled $b_0, \dots, b_3$ and the frame transition system is labelled $f_0, \dots, f_2$, both starting with b_0 respectively f_0 in the initial state. Then, the relation $R = \{(b_0, f_0), (b_0, f_1), (b_1, f_1), (b_2, f_1), (b_3, f_2)\}$ is a witness for $beh(\mathsf{Coordinator}) \sqsubseteq^{\mathrm{bb}} frm(\mathsf{Coordinator})$. Now, consider the product operator and the labels distinguished by i. Using R we obtain a witness for each transition system of the product, thus, by compositionality of blackbox refinement (Cor. 4.12), it follows that the product of the behaviours is a valid refinement of the product of the frames. Therefore, the component Coordinator is indeed correctly implemented.

The correctness proof for composite components is usually more involved. Consider, for instance, the component CashDesk as depicted in Fig. 8.3. In order to proof its correct implementation, the complete product of all subcomponent behaviours must be computed. As asynchronous connectors should be applied to be in line with the CoCoME requirements, the composition may result in an infinite-state transition system. Therefore, a manual check or a check with the MIO workbench is not feasible due to the size of the example and thus, we merely record the proof obligation that results from the correctness requirement for the component CashDesk. In order to show $beh(asm(\mathsf{CashDesk}))\rho \sqsubseteq^{\mathrm{bb}} frm(\mathsf{CashDesk})$, it is necessary to find a blackbox refinement relation such that:

$$\bigotimes_{c:C \in cmps(asm(\mathsf{CashDesk}))} \Omega(beh(c:C)\xi\alpha)\rho \sqsubseteq^{\mathrm{bb}} frm(\mathsf{CashDesk}) \ .$$

Note that the frame of CashDesk is a finite-state transition system. As already mentioned in Chap. 5, problems of this kind are called regularity problems. However, it seems that these problems are not necessarily easier to solve, compared to the case where both transition systems are infinite-state systems [**KM02**].

Frame Analysis. The analysis of component frames involves, on the one hand, analysis of properties like communication-safety for the composition of frames and, on the other hand, analysis of refinement between frames. The former is analogous to behaviour analysis, and the latter is analogous to correctness analysis. An important difference, though, is the more abstract view as given by frame specifications which makes frame analysis sometimes more amenable to manual checks. In the following we consider compatibility between the component frames for CashDeskLine and Bank, and for CashDesk and Coordinator. We conclude this section with a brief discussion on frame refinement for a simple and a more complex refinement of the component CashDeskLine.

Consider the frame specifications for CashDeskLine and Bank as given in Fig. 8.12a, 8.12c, and Fig. 8.11c. First, we assume that the components communicate synchronously and check output compatibility of $frm(\mathsf{CashDeskLine})$ and $frm(\mathsf{Bank})$. It turns out, that the frames are even strongly output compatible, due to the following informal reasoning using the given UML frame specifications. Initially, the bank component is idle and waits

according to its frame specification in Fig. 8.11c for a signal validateCard. The signal is sent by the frame of CashDeskLine in submachine SaleNorm as depicted in Fig. 8.12c before the submachine state BankComm for bank communication is entered. After the signal was sent, the submachine BankComm in turn waits for the validation result. This alternating protocol continues until BankComm reaches its final state and the frame specification of Bank reaches its initial state again. Since the bank communication is not interleaved with any other part of the component's transition systems, we conclude that $\mathit{frm}(\mathsf{CashDeskLine}) \leftrightarrow_s \mathit{frm}(\mathsf{Bank})$.

Next, assume that the components communicate asynchronously. Then, also the output queues have to be taken into account, resulting in a proof obligation of the form (assuming completely buffered frames) $\Omega(\mathit{frm}(\mathsf{CashDeskLine})) \leftrightarrow \Omega(\mathit{frm}(\mathsf{Bank}))$. Since the communication between CashDeskLine and the inventory is not interleaved with the bank communication, we consider the frames for CashDeskLine and Bank as a closed system. More precisely, we close the inventory port of CashDeskLine using a unary connector and then consider an observational equivalent frame without τ-loops. This simplified transition system mirrors the frame of Bank. As both transition systems do not diverge autonomously and both are input-separated, Thm. 5.3 is applicable and $\Omega(\mathit{frm}(\mathsf{CashDeskLine})) \leftrightarrow_w \Omega(\mathit{frm}(\mathsf{Bank}))$ follows from weak (synchronous) compatibility between the simplified frame of CashDeskLine and the frame of Bank.

As a further example, we check synchronous output compatibility of $\mathit{frm}(\mathsf{CashDesk})$ and $\mathit{frm}(\mathsf{Coordinator})$. It turns out that the frame specifications as depicted in Fig. 8.12b and Fig. 8.11a are not output compatible. After the coordinator received a message with the signal saleRegistered sent by CashDesk, the latter may proceed autonomously (using internal and output transitions) to the submachine state SaleExpr, for instance. But then, CashDesk is not ready to receive the signal expressModeEnabled that could be sent by the Coordinator now. Conceptually, such a system state is related to a timeout behaviour of the cash desk: after sale registration the component does not wait indefinitely for the decision of the coordinator. Instead, the component may internally decide to move on with the next sale process. Therefore, in order to obtain ultra-weakly output compatible frame specifications, we extend the frame of CashDesk with receive loops in the submachine states SaleExpr and SaleNorm. In this way, an expressModeEnabled signal that is delivered too late is simply ignored and the resulting frame specifications are ultra-weakly output compatible. The specifications are not yet weakly output compatible, as the initial state does not receive a signal expressModeEnabled, but the "output reachable" submachine states do.

Asynchronous output compatibility of $\mathit{frm}(\mathsf{CashDesk})$ and $\mathit{frm}(\mathsf{Coordinator})$ is verified analogously to the case of CashDeskLine and Bank above. The open ports of the component CashDesk are closed using unary connectors. Then, we observe that the obtained frame specification as well as $\mathit{frm}(\mathsf{Coordinator})$ satisfy the assumptions of Thm. 5.3 and hence $\Omega(\mathit{frm}(\mathsf{CashDesk})) \leftrightarrow_w \Omega(\mathit{frm}(\mathsf{Coordinator}))$ follows from weak (synchronous) compatibility between the simplified frame of CashDesk and the frame of Coordinator.

Our notion of blackbox refinement provides two degrees of freedom for the refined transition system. First, adding or skipping internal transitions is allowed as long as the reached states are still in refinement relation. Second, optional transitions may be dropped completely. By this means, refined transition systems may implement only parts of the specified behaviour. The frame specification, for instance, of CashDeskLine declares the transitions to its submachine states as optional (using the stereotype «opt»). As already illustrated in the context of the correctness analysis above, the PIO translation results in transition systems where mandatory, i.e., non-optional transitions are persistent. Thus, a valid blackbox refinement must only preserve non-optional transitions and hence any refinement of CashDeskLine that ignores the moves to the submachine states is a valid refinement.

For a more complex refinement problem, consider the assembly frame according to the internal structure of the composite component CashDeskLine and the frame specification of the component CashDeskLine. The connector co-cds is annotated with «async», i.e., FIFO buffered communication must be used to compute the assembly frame. Hence, in order to proof $frm(asm(\mathsf{CashDeskLine}))\rho \sqsubseteq^{\mathrm{bb}} frm(\mathsf{CashDeskLine})$, it is necessary to show that

$$(\Omega(frm(\mathsf{cashDesks{:}CashDesk})\alpha) \otimes \Omega(frm(\mathsf{coordinator{:}Coordinator})\alpha))\rho$$

is a blackbox refinement of $frm(\mathsf{CashDesk})$. For the given proof obligation we considered the singleton case, where the subcomponent cashDesks:CashDesk is declared with multiplicity 1. For the verification with an arbitrary fixed multiplicity k, an expansion of the internal structure and of the «seq-adapter» connectors as described in Chap. 7 is necessary before the frame of the assembly is computed.

CHAPTER 9

Conclusion

Before we briefly summarise the main contributions and provide prospects on possible extensions for future work, we elaborate a comparison with closely related formal software component models. The discussion is separated into a comparison of static modelling elements, support for behavioural specification, support for implementations and their description, and support for formal analysis. In this way, we relate the discussion to the categories and aspects of formal software component models as introduced in Chap. 1. We included the corresponding overview in Fig. 9.1 for convenience. The succeeding subsections comprise in each case summaries of the features of our model within the particular category.

1. Related Work

The port-based component model developed within this thesis has a focus on modelling and general behavioural analysis. We consider as relevant for a more detailed comparison only those component models from the literature having a similar objective. For example Palladio [**BKR07**] features a quite sophisticated component model. The focus, however, is on model-based performance prediction and hence the notions and concepts are strongly biased towards performance issues. Therefore, we restrict our discussion to component models that are more directly related, i.e., Sofa 2.0 [**BHP06**], GCM/Fractal [**HKR08, BABC$^+$09**], STSLib [**FR08**], and from the classical field of architectural description languages, FSP/Tracta (Darwin) [**KM06, GKC99**] and Wright [**AG97**].[1] All of these approaches feature a *component model* that defines structural entities together with a mechanism to compose them; additionally, there is an execution model that defines basic features such as the underlying communication paradigm. What qualifies these models as *software* component models is their explicit distinction of specifications and implementations of component behaviour. Finally, all of the models mentioned are *formal* software component models as they support formal reasoning usually on the level of the behavioural specifications. Figure 9.1 provides an overview of those aspects of formal software component models that we will consider more detailed within our comparison. The options given in parentheses define the values that are used in Tab. 9.1 to provide a summary of our discussion on related work.

Note that according to our definition of a software component model one might argue that Wright does not provide a *software* component model due to the missing treatment of implementations. We provide a comparison though, in order to illustrate the difference, but also the similarity with traditional ADL approaches. Another, more recently developed candidate for such a comparison is PADL [**BCD02**]. PADL and its extension as described in [**AB05**] also deals with formal modelling and verification of software architectures using component-oriented techniques. The approach extends a process algebra by architectural concepts like "architectural type" [**AB05**, Def. 2.4] that is similar to our notion of an assembly. PADL is equipped with various kinds of extension mechanisms based on a concept

[1]The given references aim at representing the most recent and most complete publication of the particular approach. Since the different aspects of a formal software component model are usually published in different articles, we provide further references within the detailed discussions below.

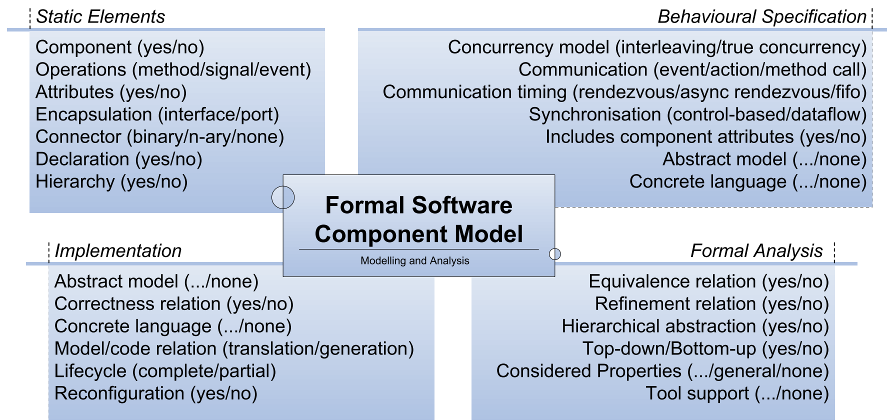


FIGURE 9.1. Aspects of formal software component models (from Chap. 1)

called "architectural type invocation". Architectural type invocations do not hide local interactions which makes a difference to our application of composite component that are used within an assembly by hiding all of its internal communications. In our view, this encapsulation is methodologically and technically quite important for building components in an hierarchical way. Such a support for hierarchical composition is also missing in Wright. In this respect it makes no difference whether we include Wright or PADL into our elaboration on related work. However, since PADL is considered more detailed in the context of an assembly analysis for synchronously communicating components, developed in Chap. 3 (Sect. 4), we decided to include Wright in here.

1.1. Static Elements. Our port-based component model was originally inspired by ROOM [**SGW94**], that later evolved to UML for Real-Time Systems [**SR98**] which in turn has been incorporated into the unified modelling language in the meantime. UML2 introduces notions of components, ports, and structured classifiers that are, not surprisingly, a perfect match with the syntactic requirements of our port-based approach to components. Hence we use UML2 as a concrete syntax for the structural specification of port-based components. Given that the UML2 shows a number of semantical variation points as well as ambiguities, see e.g. [**CGR08, FSKdR05**], we provide a metamodel for our component model that is, on the one hand, easily mappable to the corresponding notions of the UML2 and provides, on the other hand, a convenient starting point for the formalisation of static structure and behaviour of port-based components. In this sense, our semantics may also be used to remedy the ambiguities of UML2 port semantics as discussed in [**CGR08**].

As summarised in Tab. 9.1, we consider components that are encapsulated by ports with provided and required interfaces and exchange messages in the form of UML signals along binary assembly connectors. We allow the static modelling of general component (and port) attributes, but we do not allow n-ary connectors. The restriction to binary connectors allows the development of an interface theory that supports modular pairwise reasoning within component assemblies. In contrast, n-ary connectors usually demand for more global analysis techniques. We use UML declarations for ports, components, and connectors, comprising name, type, and multiplicity. Finally, we allow hierarchical system construction by encapsulation of assemblies within UML2 structured classifier using delegate connectors to link open assembly ports with the ports of the (surrounding) composite component.

SOFA 2.0 provides a technically matured component model with regard to static modelling elements. Components are encapsulated by provided and required interfaces that

TABLE 9.1. Related formal software component models

		Port-Based Components	*SOFA 2.0 [BHP06]*	*GCM/Fractal [HKR08, BABC+09]*	*STSLib [FR08]*	*FSP/Tracta [KM06,GKC99]*	*Wright [AG97, ADG98]*
Static Elements	Component	yes	yes	yes	yes	yes	yes
	Operation	signal	all	method	signal	event	event
	Component attributes	yes	yes	yes	yes	no	no
	Encapsulation	port	interface	port	interface	port	port
	Connector	binary	n-ary	n-ary	n-ary	n-ary	n-ary
	Declaration	yes	yes	yes	yes	yes	yes
	Hierarchy	yes	yes	yes	yes	yes	no
Behavioural Specification	Concurrency model	interleaving	interleaving	interleaving	interleaving	interleaving	interleaving
	Communication	actions	events	method calls	actions	actions	actions
	Comm. timing	fifo, rendezvous	rendezvous	async rendezvous	rendezvous	rendezvous	rendezvous
	Synchronisation	control-based	control-based	data-flow	control-based	control-based	control-based
	Includes attributes	no	no	no	yes	no	no
	Abstract model	PIO	Event traces	pNets	STS	LTS	LTS
	Concrete language	UML PSM	BP	JDC	STS	FSP	CSP
Implementation	Abstract model	IOTS	Event traces	pNets	STS	none	none
	Correctness relation	yes	yes	no	no	no	no
	Concrete language	UML STM, JMS	Java	Java, ProActive	Java	Java	none
	Model/code relation	translation (c)	generation (m)	generation (m)	generation (c)	translation (c)	none
	Lifecycle support	partial	complete	complete	partial	partial	partial
	Reconfiguration	no	yes	yes	yes	yes	yes
Analysis	Equivalence relation	yes (IOTS)	no	no	no	yes	yes
	Refinement relation	yes	yes	no	no	no	yes
	Hierarchical abstraction	yes	yes	no	yes	yes	no
	Top-down/Bottom-up	yes	yes	no	no	yes	no
	Considered properties	compat./safety	comm. failures	general	general	safety/liveness	compatibility
	Tool support	MIO, LASH	BP Checker	CADP toolset	CADP, STSLib	LTSA	FDR2

allow, besides signatures to be used for communication, amongst others to specify a basic kind of multiplicity and the communication style to be used for the given interface. Concerning multiplicity, "single" and "multiple bindings" are distinguished. Supported communication styles are procedure call, message-oriented communication, streaming and distributed communication via shared memory [**BP04**]. Components may refer to an arbitrary number of properties and thus modelling of component attributes is supported. Connectors may be n-ary, linking an arbitrary number of "SubcomponentInterfaceEndpoint"s. Since "Frame"s, which represent a black-box view of a component, and interface types are named entities, a notion of declaration for components and interfaces is supported. Finally, SOFA distinguishes between frames and architectures understanding the latter to represent implementations of the former. Frames may be hierarchically composed not only by using architectures but also by using again frames. Therefore, a concept of static hierarchical composition is properly supported. All of the mentioned elements refer to the presentation of the SOFA 2.0 metamodel in [**BHP06**, Appendix A].

The grid component model GCM and the software component model Fractal were initially independent developments where GCM had its source in the realm of programming support, while Fractal was developed more in the spirit of traditional ADLs, though aiming at explicit support for deployment and life cycle management of components. The most up-to-date presentation of the combined GCM/Fractal model seems to be [**HKR08**]. Fractal contributes the basis for architectural static concepts and GCM adds facilities to support group communications. The most complete description of the Fractal component model without GCM can be found in [**BCL$^+$06**]. The GCM/Fractal components represent run-time entities encapsulated by provided and required ports[2]. A port implements an interface type that specifies operations, a multiplicity, and a role which is either "client" or "server". Client ports model provided access points, while server ports model required access points. Interface types are extended by the GCM to allow for the specification of "collective interface"s either of kind multicast or gathercast [**HKR08**, p.162]. With the concepts of "primitive" and "composite bindings", there is support for the modelling of n-ary connectors. Fractal is equipped with a type system and instantiation facilities along factory components, thus a concept of component declaration is supported. The so-called "membrane" of a component features external and internal ports where the latter are not accessible from the outside. Hence, Fractal allows for proper modelling of hierarchical components.

STSLib uses a subset of KADL [**PR06**], an architectural description language for Korrigan, to define its underlying component model. Korrigan is "a formal mixed specification language which integrates [...] algebraic specifications for the static aspects and Labelled Transition Systems (LTS) for the dynamic aspects" [**PR06**, p.1744]. The approach provides an integrated model of formal component behaviours and data types. Components are encapsulated by an interface that specifies events that are emitted or received. Since events are equipped with data, the concept seems to correspond to UML signals. Connectors are not represented explicitly, but instead a more generic approach using modal logical formulas is used to relate the interfaces of different components within a so-called "communication diagram" [**PR06**, Sect. 3.3]. Thus, STSLib is considered to support a concept of n-ary connectors. The definition of composite components relies on UML composition diagrams [**FR08**, Sect. 3.5]. Therefore, it seems that a notion of declarations is supported. The static specifications shown in [**PR06**, Fig. 14–17], however, do only use type specifications similar to UML class diagrams. Composite components export events emitted or received by its subcomponents [**PR06**, pp. 1763–1764] and since only exported events are visible to the outside, hierarchical modelling is supported.

FSP (Finite State Processes) and Wright are process algebraic approaches developed in the context of architectural description languages. FSP [**KM06**] features a graphical

[2] called "interface" in [**BCL$^+$06**]

notation that is based on Darwin [**MDEK95**] and a semantic domain that is based on labelled transition systems as a mixture of the transition systems described by the process algebras CSP and CCS. The reason why we consider FSP as a related formal software component model is, on the one hand, the similarity between UML2 structured classifier and FSP structure diagrams, and, on the other hand, its emphasis on a hierarchical approach to support analysis of distributed system models using transition systems that describe (composite) component behaviours; cf. Tracta [**GKC99**].

In FSP/Tracta components are encapsulated by portals whose types are sets of event names. Events denote simple actions without taking into account parameter values explicitely [**GKC99**, p. 12]; component attributes are not considered. Since FSP allows for n-ary communication along shared actions, the approach is considered to feature n-ary connectors. Both, declarations and hierarchical system constructions are supported: the former since portals and components can be equipped with names and the latter since composite components interact via external interfaces only, effectively hiding any interaction between subcomponents. The focus for the development of the architectural description language Wright was on the modelling of connector behaviours. Even though components exist as structural entities, their detailed structural and behavioural properties are not elaborated in [**AG97**]. Components are structurally defined by a set of ports that describe logical interaction points of the respective component. A port alphabet specifies the set of event names relevant at the particular port. N-ary connectors link components using their ports. Both, component and connector types are used within declarations to specify system assemblies [**AG97**, Fig. 3]. However, arbitrary hierarchical system construction using composite components is not supported.

1.2. Behavioural Specification. Usually software component models consider components as kind of active objects that execute within their own thread of control. Note that this is also the case for ADL approaches such as FSP and Wright. Communication between components is either by (remote) method call or by message exchange. The behaviour model of port-based components was initially developed for asynchronous message exchange relying on a rendezvous (handshake) mechanism for synchronisation [**BHH$^+$06**]. In order to provide a more implementation-oriented modelling support of asynchronous message exchange, the model was extended with input queues, one for each component, which allowed to buffer received messages similar to the request queue of active objects used by the GCM/Fractal approach [**BABC$^+$09**]. However, since components are equipped with several ports representing the interaction points of a component, the execution model evolved further using one queue for each port and, finally, we switched the direction and decided to use output instead of input queues. This last decision was largely motivated by technical reasons. However, conceptually such a point of view is also taken in the context of distributed systems (cf. Sect.2), and formally, for instance, in the asynchronous π-calculus [**BPS01**, Chap. 8, Sect. 3.5].

The behavioural aspects of our model are summarised by Tab. 9.1. Likewise to the related component models, we use an interleaving semantics as our basic model of concurrency. Behaviour is formally described by receive, send and internal actions using a synchronous handshake mechanism or FIFO queues for buffered communication. Synchronisation is control-based. This common approach is in contrast to GCM/Fractal which is the only model that uses a data-flow approach to synchronisation. The latter refers to so-called future values: placeholder that are exchanged asynchronously and synchronised as soon as the corresponding value is indeed used within the client program. We do neither model component attributes nor do we consider data states of components. Formally, component behaviours are specified using input-persistent I/O-transition systems (PIO). As a concrete specification language we use a slightly extended form of UML protocol state machines. Their translation to PIOs is described in detail.

SOFA uses traces of low-level events to formally capture the behaviour and communication of components [**PV02**]. Their abstract model distinguishes whether an event is emitted, received or internal. Moreover, it is explicitly stated if the particular event is a request or a response which allows for a fine-grained modelling of both, (remote) method calls with or without return values and asynchronous message-oriented communication. SOFA applies regular-like expressions, "behaviour protocols", to specify behaviours. The semantics of a behaviour protocol is a set of event traces. The language is equipped with various operators such as sequencing, repetition, and parallel composition. The semantics of the composition operator uses a rendezvous (handshake) mechanism for the construction of interleaved event traces. Component data states are not considered, hence synchronisation is necessarily control-based.

The GCM/Fractal model is outstanding in its communication technique relying on asynchronous remote method calls with futures. A future is a placeholder variable that is either `nil` or refers to an object. If such a variable is accessed, the corresponding process is blocked until the future value is defined (not `nil`). These variables are exchanged between components to communicate return values of asynchronous method calls. A formal execution model for distributed object-oriented systems with futures is given by the ASP calculus in [**CH05**]. The execution model is discussed in the implementation part of the component model below. The abstract *behaviour* model of GCM/Fractal was initially developed for Fractal in [**BHM06**] using "pNets" (parameterised Networks of Synchronised Automata) which is a transition system based formalism that "unifies and extends the value-passing CCS of Ingolfsdottir and Lin [28], and the synchronisation networks of Arnold and Nivat [4]".[3] From the latter the basic formalism of labelled transition systems (LTS) and synchronisation vectors and networks respectively is taken, and the former gives rise to include data parameters to be used for communication events and guards. Based on an instantiation of parameters, *pNet instantiation* yields a possibly infinite LTS and defines the semantics of pNets. The formal model is elaborated and set into context of the GCM/Fractal component model in [**CM08**] and [**BABC^{+}09**]. The formalisation follows the programming model, i.e., components communicate through asynchronous method calls with futures, using a rendezvous mechanism for composition (asynchronous rendezvous, cf. [**CH05**, p.9]). The approach uses data-flow synchronisation, since components are blocked ("wait-by-necessity") as soon as the object underlying a future is first accessed. The focus so far, however, is on futures, while general component attributes are not yet considered. However, it seems that the semantics developed by Henrio et al. in [**HKR08**] could be easily extended in that respect. Quite recently the formalism was applied to sketch a semantics for transparent futures [**CHMV10**] that were not yet taken into account in the mentioned articles. Moreover, since pNets are a quite low-level formalism a concrete specification language JDC (Java Distributed Components) is currently developed by Cansado et al. [**CHM10**].

Behavioural specifications of STSLib are formally based on symbolic transition systems STS [**FR08**]. STS specify the temporal order of emitted, received and internal events taking into account guards and parameter values that are formalised along an algebraic approach to the specification of the underlying data types and their evaluation to concrete values. The transition systems are *symbolic* since parameter values are not immediately expanded as it is implicitly the case within our approach. Instead, expansion is defined by an unfolding of a STS into a configuration graph, which then again might be considered as a STS. Unfolding takes data states of components into account. The composition operator for STS uses synchronisation vectors and applies a control-based rendezvous mechanism. In some sense we could also consider the unfolded STS to provide the abstract model of behavioural specifications and the symbolic STS to provide the concrete language for behavioural specification.

[3] The cited references [28] and [4] correspond to the entries [**IL01**] and [**Arn94**] in our bibliography.

FSP and Wright are the most closely related approaches with respect to formal behavioural specification. FSP uses finite labelled transition systems and Wright relies on CSP for the specification of component, connector behaviour respectively. Being based on process algebras, the models use internal (τ) and external actions to describe behaviour and a rendezvous mechanism for the composition of components. The main difference to our approach in that respect is that we distinguish input and output actions and therefore develop a correspondingly different theory based on input-persistent I/O-transition systems (PIOs). The component models do not consider attributes. Both, FSP and CSP, are textual languages whose semantics generates labelled transition systems as a formal abstract model of behaviour.[4]

1.3. Implementation. Software component models aim at supporting explicitly a notion of component implementation. According to Lau and Wang a support of the complete life cycle (design, deployment and runtime) of a component is required.

Our approach, in particular our component model, was initially developed in the context of the architectural programming language Java/A [**Hac04**] which extends Java with architectural elements such as components, ports, and connectors. This approach provided programming support for the complete life cycle and also for dynamic reconfiguration [**BHH$^+$06**]. Since then, however, our focus shifted from programming to behavioural analysis which needs a more abstract model of component behaviours respectively component implementations. Therefore, we defined I/O-transition systems as an abstract model of the implementation of port-based components which, due to its close relation to PIOs, may also be *formally* related to specifications of component behaviour. The important difference between specification and implementation is that the former usually allows for a number of different correct implementations, while the latter may usually only be replaced by components with, for instance, observational equivalent behaviours. As a concrete language for implementations, we use a restricted simple form of UML state machines. Additionally, we describe translations to an implementation based on Java Message Service (JMS) [**Mic02**]. Both languages are close to our abstract implementation model of I/O-transition systems. Therefore, formal analysis on the level of abstract models of specification and implementation can indeed be expected to carry over to properties of the concrete implementation. Deployment is not our primary concern and therefore life cycle support is only partial: we support component design, and along our formal behavioural model also runtime concepts, but not deployment. In a similar vein, we do not consider reconfiguration.

The methodological approach concerning the difference of behavioural specification and implementation model of SOFA is similar to ours. Their approach use as an abstract implementation model again event traces. The correctness relation with respect to the behavioural specification is formally stated in [**PV02**]. As a concrete implementation language, Java is considered. The transition from the abstract model based on behaviour protocols to Java code is left unspecified. Though, the reverse direction is tackled by testing and pattern-like approaches in [**PPK06**] and [**PP09**]. SOFA provides programming support for the complete component life cycle as well as for reconfiguration issues [**BHP06**]. This support is, however, not related to their formal model of implementations and behaviours.

The implementation and execution model of GCM/Fractal is rooted in the grid component part of the approach. Within this context an object-oriented programming model and concurrent Java-based middleware implementation, ProActive, has been developed, aiming at providing programming support for general distributed systems [**BBC$^+$06**]. A formal generalisation of this approach is given by the calculus of Asynchronous Sequential Processes (ASP) [**CH05**]. Together with the Fractal component model, a quite sophisticated execution model is obtained that integrates the GCM programming model with

[4] In fact, there are different semantic models of CSP and amongst them, there is also an operational semantics generating labelled transition systems.

the management model of Fractal for the deployment and reconfiguration of hierarchical component-based systems [**CM08, BABC^{+}09**]. The resulting abstract implementation model is not yet formally related to the behavioural specifications along JDC [**CHMV10**]. Nevertheless, the reverse direction, "model generation" for ProActive implementations of GCM/Fractal components seems to be quite matured [**CM08**]. Due to the sophisticated management facilities developed within the context of Fractal, the approach provides extensive programming support for the complete component life cycle and reconfiguration as required for the dynamic evolution of a particular system.

STSLib aims at Java code generation as complete as possible. The approach does not provide a formal relation between an STS representing an implementation and a more abstract behavioural STS specification. Since STSLib is the only approach that integrates component data on the specification level, there is enough detail available to generate imperative Java code that takes both into account, the algebraic data specifications and the protocol specifications whose rendezvous-based composition is also translated to Java [**PNPR05, FPR07, FR08**]. Similar to our approach, the initial focus with STSLib respectively KADL [**PR06**] was on formal analysis of behavioural models on the transition system level. Thus, there is only partial life cycle support not considering component deployment. Dynamic reconfiguration is briefly discussed in the context of dynamic architectures [**PR06**, Sect. 3.5].

Architectural description languages intentionally focused on the specification and analysis of the software architecture of a given system. Therefore neither Darwin nor Wright provide an account on implementations or explicit models hereof. Life cycle support thus again is partial, meaning that analogously to our and the STSLib approach only design and runtime is considered but not deployment. Wright features extensions that tackle reconfiguration in the context of dynamic architectures [**ADG98**], while Darwin was initially developed with a π-calculus based semantics in order to better support the modelling and analysis of dynamic architectures [**MDEK95, MK96**]. In contrast, FSP is more in the spirit of a software component model due to its development and presentation in [**KM06**] which describes a systematic translation from FSP behavioural models to thread-based Java code.

1.4. Formal Analysis. Based on formal models of behavioural specifications and implementations, analysis aims, on the one hand, at providing early feedback during system design with regard to architectural incompatibilities and, on the other hand, by following either top-down or bottom-up development steps, to formally relate models of the same abstraction level by a refinement relation, but also of different abstraction levels by an implementation relation.

At the core of any such analysis are equivalence or refinement relations that are applied to the abstract models of component behaviour (cf. Tab. 9.1). This is particularly true for the stepwise refinement of behavioural specifications. Refinement proceeds until the specification is either concrete enough to be implemented or to be related to an abstract implementation model by some formal implementation relation. Note that this last step may involve two completely different formal languages. However, in our case of PIOs for the specification and IOTSs for the implementation, the languages relate to the same general formalism of modal I/O-automata [**LNW07**], and hence our refinement relation and our implementation relation in fact coincide. Our model explicitly supports hierarchical abstraction by hiding internal actions that stem either from local component computations or from synchronised message exchange within a composite component. Hierarchical abstraction is an important basis for the proper formal support of top-down and bottom-up approaches to system design. The former means to relate a top-level design by stepwise refinement to a decomposition of the original system into a system comprising several subcomponents. The latter means to allow for modular substitution of a component implementation in the sense that it suffices to proof a component-local correctness criterion in order to substitute without unintended effect with regard to the remaining part

of the system. Such an effect could be the invalidation of a communication property like, in our case, communication-safety ensuring for systems of port-based components that messages sent indeed will eventually be received. Compatibility and refinement analysis is supported using the MIO workbench, an Eclipse-based verification tool and editor for modal I/O automata [**May**]. In the context of the LASH toolset a C library is available, implementing algorithms and data types for a symbolic verification of systems with FIFO buffered communication based on queue content decision diagrams (QDD) [**BG99, Boi**].

SOFA's support for an analysis based on behaviour protocols is again quite closely related to our approach. In [**PV02**] an explicit refinement relation is defined taking into account the asymmetric role played by provisions and requirements of a component, i.e., by events received and events emitted. Moreover, hierarchical abstraction is supported by alphabet restriction. Formal support of top-down and bottom-up approaches to system development is essential in the mentioned article. The bottom-up direction is proven formally [**PV02**, Thm. 3.4.4] and the top-down direction is supported by a notion of correct implementation based on protocol conformance. The analysis of behaviour protocols to detect architectural incompatibilities is tackled in [**AP05**]. Considered incompatibilities comprise "bad activity", "no activity", and "divergence". Bad activity corresponds to our notion of communication-safety. No activity and divergence are related to deadlock and divergence as known from general process algebras, though taking into account the life cycle of components which allows to stop a running component, for instance to replace the particular component with a new version. There is a protocol checker [**MPK05**] that supports the verification of protocol compliance between two given behaviour protocols [**PPK06**, Sect. 3.2]. Additionally, there is a runtime checker that tests if an executing Java component obeys to its frame protocol by an integration with the Java PathFinder [**PPK06**].

According to Tab. 9.1, GCM/Fractal seems to provide only few support of formal analysis. We would like to stress that this is definitely not the case with respect to general properties. For the comparison of analysis techniques, however, we were particularly interested in formal support of *component-related* properties as suggested by the characteristics for CBD derived in Sect. 1. In that respect, there is a need for specific analysis techniques that are not yet provided by the GCM/Fractal approach [**CHMV10**, Sect. 4]. In contrast, the support for the verification of general properties of their pNet models is quite extensive using the CADP toolset [**INR**] that allows to check a large number of preorders and equivalence relations, and also allows to model-check, for instance, μ-calculus formulas [**CM08**, Sect. 4.6].

With STSLib the situation is similar to GCM/Fractal. There is extensive support for general verification issues including approaches that involve theorem provers, but no dedicated support for an analysis in a component-based setting [**PR06**, Sect. 4.3]. STS can be translated to LOTOS specifications paving the way to use CADP for general verification issues. An STSLib API is currently developed aiming at providing more direct verification support [**FR08**, Sect. 4.2]. The STS formalism features internal actions and hence we consider STSLib to provide means of hierarchical abstraction.

FSP/Tracta use observational equivalence for a compositional reachability analysis of labelled transition systems [**GKC99**]. Even though no refinement relation is employed, we consider the approach to provide support for top-down and bottom-up design due to its systematic approach to behavioural analysis in Tracta. In particular, the verification of safety and liveness properties in hierarchically structured systems is supported in a quite intuitive way. Properties are specified using finite deterministic LTS or Büchi automata respectively and may relate to actions of subsystems at an arbitrary hierarchical level [**GKC99**, Sect. 3] of the given system architecture. Verification is supported by the Labelled Transition System Analyser LTSA that allows also LTL model checking [**MKC$^+$**].

Wright is based on CSP and thus inherits its equivalence and refinement relations as well as support for the verification of general properties using the FDR model checker

[**For**]. Additionally a compatibility relation based on CSP refinement is defined for connector behaviours and port behaviours (of attached components) that is used to check a notion of architectural compatibility [**AG97**]. Since Wright does not consider hierarchical systems, there is neither support for hierarchical abstraction nor support for top-down and bottom-up design approaches in component-based development.

2. Evaluation and Prospects

An integrated formal software component model with support for synchronous and asynchronous message exchange as prevalent, for instance, in distributed systems was developed. In particular, the difference of abstract models for component implementation and component specification was elaborated, resulting in an IOTS-based theory of component implementations and a PIO-based theory of component specifications. The theories use a compositional notion of refinement that preserves specific properties (different notions of communication compatibility) and general properties (safety properties). A definition of correct implementation established a formal link between implementations and specifications. On this basis, support for the characteristics of component-based development was stated formally. We discussed a detailed translation to CFSM systems in order to support the verification of systems with asynchronous communication. We used the UML2 as a concrete specification language for static as well as dynamic aspects and provided a systematic translation to the more abstract level of transition systems. The CoCoME was used to illustrate the features of our component model, to exemplify the translation of UML state machines and to discuss proof obligations that might arise in a formal analysis of the given specifications.

Evaluation. An integrated formal software component model provides a complete picture of static structure, behaviour specification and behaviour implementation. In particular, the separated abstract models allow a separated development of corresponding theories and verification approaches, while implementation correctness provides the formal link that is required to obtain an integrated model. Thus, we believe that the model developed within this thesis also provides a useful and detailed view of the general constituents of a formal software component model.

Frame specifications differ from behaviour specifications, amongst others, by using only τ actions for the modelling of internal choice. We believe that such a restriction helps to focus on the externally visible behaviour while specifying dynamical aspects. In contrast, a distinguished local alphabet is available for the description of component behaviours and thus helps to describe the implementation behaviour of components in a direct way.

For practical applicability, it is important that the abstract models are conceptually closely related to the respective concrete domains. For example, IOTSs provide a direct match with message-oriented systems that communicate by rendezvous or FIFO-buffered message exchange. However, they do not provide a direct match with systems that communicate by method calls.

Ports turned out to be an important structural modelling element. The UML concept of provided and required interfaces of a port provides an appropriate means for the syntactical definition of the communication alphabet at a particular interaction point of a component. However, the role of port protocols for modular verification seems to be limited, as the required assumptions on the relation between port protocol and component behaviour are, at least for the verification of our notions of communication compatibility, rather strong.

Limitations and Future work. Some of the shortcomings of the developed component model directly point to potential extensions and improvements. First, life cycle support is only partial. According to [**LW07**] a complete life cycle support including deployment is important for practical software component models. Thus, concepts for component-based

deployment as well as support for programming, installation and update of component types should be included. Eventually, this touches also conceptual, formal, and practical support for reconfiguration. Such a support calls for a precise distinction between types and instances which directly relates to a second limitation. We interpreted declarations of components and ports as architectural templates for different instantiation possibilities defined by the given multiplicities. In this way, we considered in fact the instance but not the type level by our formal models of dynamic system specifications. We believe that proper support of arbitrary multiplicities could be integrated with a support for reconfiguration. Third, group communication was not considered. However, broadcast and multicast communication is directly related to the publish/subscriber pattern that is, besides point-to-point communication, the second prevalent communication mechanism in message-oriented systems. Therefore, adding a concept of n-ary connectors together with an appropriate theory for group communication seems to be a valuable effort.

Furthermore, tool support for verification is incomplete but indispensable for larger case studies and practical application. This is particularly true for the formal analysis of systems with asynchronous communication. Here, we believe that the CFSM-based symbolic verification approach with QDDs is quite promising. As algorithms for the construction of QDDs exist, an integration with an existing verification tool could be a feasible next step. For instance, the MIO workbench [**May**] already supports the verification of weak output compatibility and blackbox refinement. Thus, it would be convenient to integrate also support for the verification of systems with buffered communication. Another candidate for a corresponding extension is the LTSA tool [**MKC**$^+$] which is a quite useful tool for the verification of labelled transition systems.

Besides, there are extensions that are beyond a mere improvement of the existing model. For instance, in order to develop an approach for asynchronously communicating components with data states, all categories that constitute a formal software component model need to be revisited. First of all, it must be clarified which kind of operational interaction model is appropriate: signals or remote method calls. Next, an appropriate interpretation of data-related behavioural specifications such as pre- and postconditions of operations in an asynchronous context is still an open research question. Building on such an interpretation, it would be interesting, on the level of implementations, to develop a design-by-contract approach for a concrete concurrent programming language. Finally, formal analysis of data-related specifications is, similar to the analysis of systems with FIFO-buffered communication, related to the verification of infinite-state systems and hence a research issue on its own.

Bibliography

[AB05] Alessandro Aldini and Marco Bernardo. On the Usability of Process Algebra: An Architectural View. *Theor. Comput. Sci.*, 335(2-3):281–329, 2005.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 187–197. ACM, 2002.

[ADG98] Robert Allen, Remi Douence, and David Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon and Portugal, March 1998.

[AG97] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[AP05] Jirí Adámek and Frantisek Plasil. Component Composition Errors and Update Atomicity: Static Analysis. *J. Softw. Maint. Evol.-R.*, 17(5):363–377, 2005.

[Arn94] André Arnold. *Finite Transition Systems. Semantics of Communicating Systems*. Prentice-Hall, Englewood, 1994.

[BABC+09] Tomás Barros, Rabéa Ameur-Boulifa, Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Behavioural Models for Distributed Fractal Components. *Annales des Télécommunications*, 64(1–2):25–43, 2009.

[BBC+06] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming and Deploying and Composing and for the Grid. Springer-Verlag, January 2006.

[BCD02] Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. Architecting Families of Software Systems with Process Algebras. *ACM Trans. Softw. Eng. Methodol.*, 11(4):386–426, 2002.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL Component Model and its Support in Java. *Softw. and Pract. Exper.*, 36(11–12):1257–1284, 2006.

[BFH+07] Buschmann, Frank, Henney, Kevlin, Schmidt, and Douglas C. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. Wiley, June 2007.

[BG99] Bernard Boigelot and Patrice Godefroid. Symbolic Verification of Communication Protocols with Infinite State Spaces using QDDs. *Formal Methods in System Design*, 14(3):237–255, 1999.

[BGWW97] Bernard Boigelot, Patrice Godefroid, Bernard Willems, and Pierre Wolper. The Power of QDDs (Extended Abstract). In Pascal Van Hentenryck, editor, *Proceedings of the 4th International Symposium on Static Analysis (SAS'97)*, volume 1302 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 1997.

[BHH+06] Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. A Component Model for Architectural Programming. *Elect. Notes Theo. Comp. Sci. (FACS'05)*, 160:75–96, 2006.

[BHM06] Tomas Barros, Ludovic Henrio, and Eric Madelaine. Verification of Distributed Hierarchical Components. *Elect. Notes Theo. Comp. Sci. (FACS'05)*, 160:41–55, 2006.

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *4th International Conference on Software Engineering Research, Management and Applications (SERA'06)*, pages 40–48. IEEE Computer Society, 2006.

[BKL08] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of Model Checking*. MIT Press, 2008.

[BKR07] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-Based Performance Prediction with the Palladio Component Model. In Vittorio Cortellessa, Sebastián Uchitel, and Daniel Yankelevich, editors, *Proceedings of the 6th International Workshop on Software and Performance (WOSP)*, pages 54–65. ACM, 2007.

[BMSH10] Sebastian S. Bauer, Philip Mayer, Andreas Schroeder, and Rolf Hennicker. On Weak Modal Compatibility and Refinement and and the MIO Workbench. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6015 of *Lect. Notes Comp. Sci.*, pages 175–189. Springer, 2010.

[Boi] Bernard Boigelot. The LASH toolset. http://www.montefiore.ulg.ac.be/~boigelot/research/index.html#lash (2010-02-21).

[BOR04] Steffen Becker, Sven Overhage, and Ralf Reussner. Classifying Software Component Interoperability Errors to Support Component Adaption. In Ivica Crnkovic, Judith Stafford, Heinz Schmidt, and Kurt Wallnau, editors, *Proceedings of the 7th International Symposium on Component-Based Software Engineering*, volume 3054 of *Lect. Notes Comp. Sci.*, pages 68–83. Springer, 2004.

[BP04] Tomas Bures and Frantisek Plasil. Communication Style Driven Connector Configurations. In *1st International Conference on Software Engineering Research and Applications (SERA'03), Selected Revised Papers*, volume 3026 of *Lect. Notes Comp. Sci.*, pages 102–116. Springer, 2004.

[BPS01] Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science B.V., 2001.

[BZ83] Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.

[CF05] Gérard Cécé and Alain Finkel. Verification of Programs with Half-Duplex Communication. *Information and Computation*, 202(2):166–190, 2005.

[CFN03] Cyril Carrez, Alessandro Fantechi, and Elie Najm. Behavioural Contracts for a Sound Composition of Components. In *Proceedings of the 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'03)*, volume 2767 of *Lect. Notes Comp. Sci.*, pages 111–126. Springer, 2003.

[CGR08] Arnaud Cuccuru, Sébastien Gérard, and Ansgar Radermacher. Meaningful Composite Structures. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301 of *Lect. Notes Comp. Sci.*, pages 828–842. Springer, 2008.

[CH05] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects*. Springer-Verlag, 2005.

[CHM10] Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Transparent First-class Futures and Distributed Components. *Electr. Notes Theor. Comput. Sci.*, 260:155–171, 2010.

[CHMV10] Antonio Cansado, Ludovic Henrio, Eric Madelaine, and Pablo Valenzuela. Unifying Architectural and Behavioural Specifications of Distributed Components. *Electr. Notes Theor. Comput. Sci.*, 260:25–45, 2010.

[CK96] Shing Chi Cheung and Jeff Kramer. Context Constraints for Compositional Reachability Analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(4):334–377, 1996.

[CM08] Antonio Cansado and Eric Madelaine. Specification and Verification for Grid Component-Based Applications: From Models to Tools. In de Boer et al. [**dBBM09**], pages 180–203.

[Crn03] Ivica Crnković. Component-Based Software Engineering - New Challenges in Software Development. *Journal of Computing and Information Technology*, 11(3):151–161, September 2003.

[CSSW04] Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau. 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction. *SIGSOFT Softw. Eng. Notes*, 29(3):1–7, 2004.

[CVZ06] Ivana Cerná, Pavlína Vařeková, and Barbora Zimmerova. Component Substitutability via Equivalencies of Component-Interaction Automata. In *Proc. Wsh. Formal Aspects of Component Software (FACS'06)*, pages 115–130, Praha, 2006. UNU-IIST Technical Report 344.

[dAH01a] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 9th Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[dAH01b] Luca de Alfaro and Thomas A. Henzinger. Interface Theories for Component-Based Design. In *EMSOFT: Embedded Software*, volume 2211 of *Lect. Notes Comp. Sci.*, pages 148–165. Springer, 2001.

[dAH05] Luca de Alfaro and Thomas A. Henzinger. Interface-based Design. In Manfred Broy, Johannes Grünbauer, David Harel, and C. A. R. Hoare, editors, *Engineering Theories of Software-intensive Systems*, volume 195 of *NATO Science Series: Mathematics and Physics and and Chemistry*, pages 83–104. Springer, 2005.

[dBBM09] Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelain, editors. *7th International Symposium on Formal Methods for Components and Objects (FMCO'08), Sophia Antipolis, France, October 21-23, 2008, Revised Lectures*, volume 5751 of *Lect. Notes Comp. Sci.* Springer, 2009.

[DOS09] Francisco Durán, Meriem Ouederni, and Gwen Salaün. Checking Protocol Compatibility using Maude. *Electr. Notes Theor. Comput. Sci.*, 255:65–81, 2009.

[EDM08] Shahram Esmaeilsabzali, Nancy A. Day, and Farhad Mavaddat. Interface Automata with Complex Actions: Limiting Interleaving in Interface Automata. *Fundam. Inform.*, 82(4):465–512, 2008.

[Elo91] Jaana Eloranta. Minimizing the Number of Transitions with Respect to Observation Equivalence. *BIT*, 31(4):576–590, 1991.

[FBS05] Xiang Fu, Tevfik Bultan, and Jianwen Su. Synchronizability of Conversations among Web Services. *IEEE Trans. Software Eng.*, 31(12):1042–1055, 2005.

[For] Formal Systems (Europe) Limited. FDR2.83 academic use release. http://www.fsel.com (2010-03-29).

[FPR07] Fabrício Fernandes, Robin Passama, and Jean-Claude Royer. Components with Symbolic Transition Systems: A Java Implementation of Rendezvous. In Alistair A. McEwan, Steve A. Schneider, Wilson Ifill, and Peter H. Welch, editors, *CPA*, volume 65 of *Concurrent Systems Engineering Series*, pages 89–107. IOS Press, 2007.

[FR08] Fabrício Fernandes and Jean-Claude Royer. The STSLib Project: Towards a Formal Component Model Based on STS. *Electr. Notes Theor. Comput. Sci.*, 215:131–149, 2008.

[FSKdR05] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 New Unclarities in the Semantics of UML 2.0 State Machines. In *7th International Conference on Formal Engineering Methods (ICFEM'05)*, volume 3785 of *Lect. Notes Comp. Sci.*, pages 52–65. Springer, 2005.

[GGMC+07] Gregor Gössler, Susanne Graf, Mila Majster-Cederbaum, Moritz Martens, and Joseph Sifakis. Ensuring Properties of Interaction Systems by Construction. In *Program Analysis and Compilation and Theory and Practice — Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *Lect. Notes Comp. Sci.*, pages 201–224. Springer, 2007.

[Gia99] Dimitra Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College of Science and Technology and Medicine and University of London and Department of Computing, 1999.

[GKC99] Dimitra Giannakopoulou, Jeff Kramer, and Shing Chi Cheung. Behaviour Analysis of Distributed Systems Using the Tracta Approach. *Automated Software Eng.*, 6(1):7–35, 1999.

[GMY84] M.G. Gouda, E.G. Manning, and Y.T. Yu. On the Progress of Communication between two Finite State Machines. *Inf. Control*, 63(3):200–216, 1984.

[GS05] Gregor Gössler and Joseph Sifakis. Composition for Component-based Modeling. *Sci. Comp. Prog.*, 55(1-3):161–183, 2005.

[GTBF03] Holger Giese, Matthias Tichy, Sven Burmester, and Stephan Flake. Towards the Compositional Verification of Real-Time UML Designs. *SIGSOFT Softw. Eng. Notes*, 28(5):38–47, 2003.

[Hac04] Florian Hacklinger. Java/A - Taking Components into Java. In *Proceedings of the 13th Conference on Intelligent and Adaptive Systems and Software Engineering*, pages 163–168. ISCA, 2004.

[HJK08] Rolf Hennicker, Stephan Janisch, and Alexander Knapp. On the Observable Behaviour of Composite Components. In Carlos Canal and Corina Pasareanu, editors, *Proc. 5th Int. Wsh. Formal Aspects of Component Software (FACS'08)*, 2008.

[HJK09] Rolf Hennicker, Stephan Janisch, and Alexander Knapp. Refinement of Components in Connection-Safe Assemblies with Synchronous and Asynchronous Communication. In C. Choppy and O. Sokolsky, editors, *Proc. of 15th Int. Monterey Wsh., Foundations of Computer Software, Future Trends and Techniques for Development*, volume 6028 of *Lect. Notes Comp. Sci.*, pages 154–180. Springer, 2009.

[HJS01] Michael Huth, Radha Jagadeesan, and David A. Schmidt. Modal Transition Systems: A Foundation for Three-Valued Program Analysis. In David Sands, editor, *10th European Symposium on Programming Languages and Systems (ESOP'01)*, volume 2028 of *Lect. Notes Comp. Sci.*, pages 155–169. Springer, 2001.

[HKR08] Ludovic Henrio, Florian Kammüller, and Marcela Rivera. An Asynchronous Distributed Component Model and its Semantics. In de Boer et al. [**dBBM09**], pages 159–179.

[HKW+07] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. CoCoME — The Common Component Modeling Example. In Rausch et al. [**RRMP08**], pages 16–53.

[IL01] A. Ingolfsdottir and H. Lin. *A Symbolic Approach to Value-Passing Processes*, chapter 7, pages 427–478. In Bergstra et al. [**BPS01**], 2001.

[INR] INRIA (Grenoble/Rhone-Alpes). Construction and Analysis of Distributed Processes. Software Tools for Designing Reliable Protocols and Systems. `http://www.inrialpes.fr/vasy/cadp` (2010-03-29).

[IU01] Paola Inverardi and Sebastian Uchitel. Proving Deadlock Freedom in Component-Based Programming. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (FASE'01)*, pages 60–75, London and UK, 2001. Springer.

[Kaa92] Marinus Frans Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit te Amsterdam, 1992.

[KJH+08a] Alexander Knapp, Stephan Janisch, Rolf Hennicker, Allan Clark, Stephen Gilmore, Florian Hacklinger, Hubert Baumeister, and Martin Wirsing. Modelling the CoCoME with the Java/A Component Model. In Rausch et al. [**RRMP08**], pages 207–237.

[KJH+08b] Alexander Knapp, Stephan Janisch, Rolf Hennicker, Allan Clark, Stephen Gilmore, Florian Hacklinger, Hubert Baumeister, and Martin Wirsing. Modelling the CoCoME with the Java/A Component Model (Extended Version). `http://www.pst.ifi.lmu.de/Research/current-projects/cocome/cocome-extended.pdf/view` (2010-04-06), 2008.

[KM99] Jeff Kramer and Jeff Magee. *Concurrency. State Models and Java Programs*. Worldwide Series in Computer Science. John Wiley and Sons, March 1999.

[KM02] Antonín Kucera and Richard Mayr. Simulation Preorder over Simple Process Algebras. *Information and Computation*, 173(2):184–198, 2002.

[KM06] Jeff Kramer and Jeff Magee. *Concurrency. State Models and Java Programs, 2nd Edition*. Worldwide Series in Computer Science. John Wiley and Sons, April 2006.

[Kuc99] Antonín Kucera. On Finite Representations of Infinite-State Behaviours. *Information Processing Letters*, 70(1):23–30, 1999.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice-Hall, Englewood Cliffs, 2004.

[LMW04] Stefan Leue, Richard Mayr, and Wei Wei. A Scalable Incomplete Test for the Boundedness of UML RT Models. In Kurt Jensen and Andreas Podelski, editors, *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lect. Notes Comp. Sci.*, pages 327–341. Springer, 2004.

[LNW07] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal I/O Automata for Interface and Product Line Theories. In Rocco De Nicola, editor, *16th European Symposium on Programming Languages and Systems (ESOP'07)*, volume 4421 of *Lect. Notes Comp. Sci.*, pages 64–79. Springer, 2007.

[LT88] Kim Guldstrand Larsen and Bent Thomsen. A Modal Process Logic. In *Proceedings and 3rd Annual Symposium on Logic in Computer Science (LICS)*, pages 203–210. IEEE Computer Society, 1988.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Trans. Software Eng.*, 33(10):709–724, 2007.

[May] Philip Mayer. The MIO Workbench. `http://www.miowb.net` (2010-03-29).

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In Wilhelm Schäfer and Pere Botella, editors, *5th European Software Engineering Conference (ESEC'95)*, volume 989 of *Lect. Notes Comp. Sci.*, pages 137–153. Springer, 1995.

[Mic] Microsoft. .Net. `http://www.microsoft.com/net` (2010-03-22).

[Mic02] Sun Microsystems. Java Message Service Specification (V1.1). Internet, April 2002.

[Mic07] Sun Microsystems. Sun Java System Message Queue 4.1 Technical Overview. Technical report, Sun Microsystems, 2007.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall and Inc., Upper Saddle River and NJ and USA, 1989.

[MK96] Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, 1996.

[MKC$^+$] Jeff Magee, Jeff Kramer, Robert Chatley, Sebastian Uchitel, and Howard Foster. LTSA – Labelled Transition System Analyser. `http://www.doc.ic.ac.uk/ltsa` (2010-03-29).

[MPK05] M. Mach, F. Plasil, and J. Kofron. Behavior Protocol Verification: Fighting State Explosion. *International Journal of Computer and Information Science*, 6(1):22–30, 2005.

[MPR04] Olivier Maréchal, Pascal Poizat, and Jean-Claude Royer. Checking Asynchronously Communicating Components Using Symbolic Transition Systems. In *Proceedings of OTM Confederated International Conferences, CoopIS, DOA, and ODBASE, Part II*, volume 3291 of *Lect. Notes Comp. Sci.*, pages 1502–1519. Springer, 2004.

[NR68] P. Naur and B. Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee*. Brussels and Scientific Affairs Division and NATO (1969), 1968. Garmisch, Germany, 7-11 Oct.

[NV90] Rocco De Nicola and Frits W. Vaandrager. Action versus State based Logics for Transition Systems. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *Lect. Notes Comp. Sci.*, pages 407–419. Springer, 1990.

[OMG] Object Management Group OMG. Corba. `http://www.corba.org` (2010-03-22).

[OMG09] Object Management Group OMG. Unified Modeling Language Superstructure Version 2.2. `http://www.omg.org/spec/UML/2.2/Superstructure/PDF` (2010-30-30), 2009.

[PNPR05] Sebastian Pavel, Jacques Noyé, Pascal Poizat, and Jean-Claude Royer. A Java Implementation of a Component Model with Explicit Symbolic Protocols. In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Software Composition*, volume 3628 of *Lect. Notes Comp. Sci.*, pages 115–124. Springer, 2005.

[PP09] Tomás Poch and Frantisek Plasil. Extracting Behavior Specification of Components in Legacy Applications. In Grace A. Lewis, Iman Poernomo, and Christine Hofmeister, editors, *Proceedings of the 12th International Symposium on Component-Based Software Engineering (CBSE'09)*, volume 5582 of *Lect. Notes Comp. Sci.*, pages 87–103. Springer, 2009.

[PPK06] Pavel Parizek, Frantisek Plasil, and Jan Kofron. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. In *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, pages 133–141. IEEE Computer Society, 2006.

[PR06] Pascal Poizat and Jean-Claude Royer. A Formal Architectural Description Language based on Symbolic Transition Systems and Temporal Logic. *Journal of Universal Computer Science*, 12(12):1741–1782, 2006.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[RJB05] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual, 2nd edition*. Pearson Education and Inc, 2005.

[RRMP08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lect. Notes Comp. Sci.* Springer, 2008.

[SC00] João Costa Seco and Luis Caires. A Basic Model of Typed Components. In Elisa Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lect. Notes Comp. Sci.*, pages 108–128. Springer, 2000.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, Inc., 1994.

[Som06] Ian Sommerville. *Software Engineering, 8th edition*. Pearson Education, 2006.

[SR98] Bran Selic and Jim Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical report, ObjectTime Limited and Rational Software Corporation, March 1998.

[TGG+05] Tobin Titus, Syed Fahad Gilani, Mike Gillespie, James Hart, Benny K. Mathew, Andy Olsen, David Curran, Jon Pinnock, Robin Pars, Fabio Claudio Ferracchiati, Sandra Gopikrishna, Tejaswi Redkar, and Srinivasa Sivakumar. *Pro .NET 1.1 Remoting, Reflection, and Threading*. Apress, 2005.

[TM07] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems – Principles and Paradigms, Second Edition*. Pearson International Edition, 2007.

[Tuc04] Allen B. Tucker, editor. *Computer Science Handbook, Second Edition*. Chapman & Hall/CRC, 2004.

[vB78] Gregor von Bochmann. Finite State Description of Communication Protocols. *Computer Networks*, 2:361–372, 1978.