Refinement of Components in Connection-Safe Assemblies with Synchronous and Asynchronous Communication[⋆]

Rolf Hennicker[1], Stephan Janisch[1], and Alexander Knapp[2]

[1] Institut für Informatik, Ludwig-Maximilians-Universität München
{hennicker,janisch}@pst.ifi.lmu.de
[2] Institut für Informatik, Universität Augsburg
knapp@informatik.uni-augsburg.de

Abstract. Components are strongly encapsulated behaviours which interact with the environment by exchanging messages. Interaction may, amongst others, follow a synchronous rendezvous mechanism for message exchange or an asynchronous paradigm where sending and handling a message happens at different points in time. We extend our previously defined component model by integrating synchronous and asynchronous communication. As the formal background we use I/O-transition systems and consider asynchronous communication with fifo-ordered message buffers. We identify compatibility properties that should be satisfied when components communicate along synchronous and asynchronous connectors. As a first result we show that synchronous compatibility is a sufficient condition to ensure buffered compatibility in asynchronous communications. We introduce the notion of connection-safe assemblies which requires compatibility of both kinds of communication. We define a refinement relation and show its compositionality with respect to synchronous and asynchronous connectors in connection-safe assemblies. Finally, we provide results showing the preservation of connection-safety under component refinement.

1 Introduction

Structuring of large-scale software systems in terms of components and their interconnections is nowadays a standard in software development. Components can be characterised by the ability to encapsulate internal behaviour and to provide well-defined access points (often called ports) to the outside. This supports the construction of component assemblies, by connecting components via their ports, and the substitutability of components by relying only on their observable behaviour. Various software component models have been proposed; for an overview see [1] and for a comparison in the context of a "common component modelling example" (CoCoME) see [2].

The current paper sets out from our component model provided in [3] which has been equipped with a precise formalisation of the structural and behavioural aspects of components. This model distinguishes between simple (i.e., basic) components, component

[⋆] This research has been partially supported by the GLOWA-Danube project (01LW0602A2) sponsored by the German Federal Ministry of Education and Research.

assemblies, i.e., network structures of components connected via their ports, and composite components which encapsulate assemblies. For the formal representation of behaviours we use I/O-transition systems which are based on interface automata [4]. Explicit I/O-labellings allow us to distinguish between input, output and internal actions, which can be hidden to compute (derive) the observable behaviour of a component. The behaviour of an assembly is also an I/O-transition system which is derived from the composition of the observable behaviours of the components connected within the assembly. We assume that connections are always binary. For the computation of an assembly behaviour, the communication behaviour between connected components plays, of course, a central role. In [3] we have considered the case of synchronous communication, where communication is achieved by a rendezvous mechanism such that the sender and the receiver of a message synchronise on message exchange. This case is also considered in most other approaches like in ADLs such as Wright, Darwin or PADL [5]; but also software component models such as SOFA 2.0 and Fractal, both using behaviour protocols, or CoIn [6] with component interaction automata follow the synchronous communication scheme.

The first goal of this paper is to extend our component model in [3] by taking into account synchronous *and* asynchronous communication where a fifo-buffering mechanism is used when a message is sent along an asynchronous connector. Even though asynchronous communication with buffering is frequently used in practice, in particular in the context of distributed systems, its semantic properties are often neglected when it comes to behavioural analysis. One contribution of this paper lies in the rigorous formal treatment of asynchronous communication in our component model. For this purpose we provide a detailed definition for the computation of an assembly behaviour on the basis of synchronous and asynchronous connectors integrating explicit buffer behaviour and component behaviour on the level of I/O-transition systems. The resulting systems resemble communicating finite state machines (CFSM) with unbounded fifo-channels [7], and, in fact, we expect the theoretical results from the broad literature on the verification of CFSM systems, e.g., testing approaches to the unboundedness problem [8], to be more or less readily applicable to our semantics. In the realm of software component models, Maréchal et al. [9] give an approach to the analysis of components with asynchronous communication for Korrigan [10] based on symbolic transition systems. Asynchronous communication is explicitly taken into account by an integration of mailboxes for messages received but not yet processed; this work aims at developing algorithms for mailbox analysis, such as boundedness checks for mailboxes with (fifo) or without order (dictionary). Other approaches such as SOFA 2.0, Fractal, or Java/A [11] usually cope with asynchronous communication only in terms of an implementation which is not directly applicable to formal analysis. More recently, a Fractal extension [12] aims at a formal semantics for the behavioural modelling of distributed systems based on pNets [13]. However, the latter focuses on remote method calls as a mechanism for asynchronous communication.

None of the mentioned approaches provides equivalence or refinement relations for component behaviour, which directly leads to our second important goal, focusing on the study of component refinement and the investigation of properties concerning the communication behaviour of components (in assemblies) to be preserved under refinements. To consider component refinement we must compare the observable behaviours of components which are given by I/O-transition systems. Since I/O-transition systems are based on the interface automata of de Alfaro and Henzinger [4], we can reuse their quite appealing ideas for refinement defined in terms of an alternating simulation relation. Essentially, any input of an abstract behaviour must be accepted as an input of the concrete behaviour (in related states) and, conversely, any output produced by a concrete behaviour must be producible as an output of the abstract behaviour (in related states). In principle, we adopt this idea but we suggest two extensions. First, we distinguish between internal actions and the invisible action τ, because internal actions naturally appear to express communications between components (which are neither input nor output actions) but which can not be abstracted away since we are interested in communication behaviours within component assemblies. Secondly, in contrast to [4], we require additional conditions for refinement, similar to the requirements of stuck-free conformance as defined in [14], to ensure that reactiveness of abstract behaviours is also valid in concrete behaviours.

On the basis of our refinement notion we study relationships between component refinement and communication behaviour exposed by component assemblies. For this purpose, we have to distinguish between synchronous and asynchronous communication. In the synchronous case two components (more precisely, observable component behaviours) are called *(synchronously) compatible* if any output issued by one component meets the other component in a state where this output will be accepted as an input, and vice versa. This notion, however, cannot be directly applied for asynchronously communicating components since then there is a delay between the action of sending a message to a buffer and the action of the target component taking the message from the buffer. Hence, in the asynchronous case, the interesting question is whether any message sent by one component will eventually be taken from the buffer by the other component for further processing. If this property is satisfied the two components are called *buffered compatible*. In our component model we allow both, synchronous and asynchronous connectors and we call a component assembly *connection-safe* if compatibility is ensured, for each kind of connectors, in any global system run.

As one major result we draw a connection between synchronous and asynchronous communication behaviour and show, by extending a theorem of [15], that synchronously compatible components are also buffered compatible if they are put in an asynchronous environment. From the practical point of view this result is rather relevant because checking buffered compatibility directly may soon become unmanageable while checks for synchronous compatibility are usually much easier. Our main theorems show that for closed assemblies with two connected components the following holds: First, the refinement relation is compositional for synchronous as well as for asynchronous component connectors and, secondly, the properties of synchronous and buffered compatibility, i.e. connection safety, are preserved by component refinement. In particular, this implies substitutability of components within an assembly by preserving connection-safety.

The paper is organised as follows: The basic definitions and facts for I/O-transition systems needed for this study are summarised in Sect. 2. In Sect. 3 we present the extension of our component metamodel to synchronous and asynchronous connectors and introduce a running example. In Sect. 4, we provide a detailed account on the corresponding extension of the algebraic formalisation of our component model, integrating the definition of assembly behaviour with synchronous and asynchronous connectors. Sections 5 and 6 constitute our main results. In Sect. 5, the notion of connection-safe assembly is introduced and the relationship between the different compatibility notions is analysed. In Sect. 6 we define the refinement relation for I/O-transition systems and components and we provide the compositionality results for synchronous and asynchronous connectors. Finally, we give some concluding remarks in Sect. 7.

2 I/O-Transition Systems

We use I/O-transition systems to describe behaviours of ports, components, and assemblies with their provided (input) and required (output) operations as well as their internal actions. Our definition of I/O-transition system is similar to the notion of interface automata of de Alfaro and Henzinger [4]. However, we distinguish between internal actions and the *invisible* (or *silent*) action τ, because we are also interested in behaviours where internal actions should not be abstracted. For instance, we will focus on assembly behaviours of connected components where interactions between components are internal (because they are neither input nor output). Then we are interested in properties of interaction behaviours which can only be studied if internal actions are not abstracted. But, of course, when climbing up the hierarchal structure of components then the behaviour of a composite component, which encapsulates an assembly, will be obtained by abstracting the internal interactions to τ. In the following we summarise our notions for I/O-transition systems presented in [3] which will be used hereafter.

An *I/O-labelling*, *iol* for short, $L = (I, O, T)$ 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 $I \cup O \cup T$. An *I/O-transition system*, *iots* for short, $A = (L, S, s_0, \Delta)$ is given by an iol L, a set of *states* S, an *initial state* $s_0 \in S$ and a *transition relation* $\Delta \subseteq S \times (\bigcup L \cup \{\tau\}) \times S$ (with $\tau \notin \bigcup L$). We write $L(A)$ for the iol of A.

2.1 Operators on I/O-Transition Systems

For deriving behaviours in our component framework we will use the following operators on iotss: hiding, relabelling and the formation of products. Hiding and relabelling on iotss are generalisations of the usual operators used in process algebras (see, e.g., [16,5], and the product is defined in accordance with the product of interface automata [4].

Hiding. Hiding is used to turn a subset of the labels of an iots into the invisible action τ. Formally, 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 $\Delta/H = \{(s, \tau, s') \mid (s, a, s') \in \Delta \wedge a \in H\} \cup \{(s, a, s') \mid (s, a, s') \in \Delta \wedge a \notin H\}$.

In some 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 with $L(A) = (I, O, T)$, we write $A\xi$ for A/T.

Relabelling. A relabelling is used for renaming labels and for changing the kind of labels. Formally, 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 from $\bigcup L$ to $\bigcup L'$ for which we also write ρ. 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 $\Delta\rho = \{(s, \rho(l), s') \mid (s, l, s') \in \Delta \wedge l \in \bigcup L\} \cup \{(s, \tau, s') \mid (s, \tau, s') \in \Delta\}$.

Given two relabellings $\rho_1 : L \to L'$ and $\rho_2 : L \to L'$, we define their union by $\rho_1 \cup \rho_2 : L \to L'$ with $(\rho_1 \cup \rho_2)(l) = \rho_1(l)$ if $\rho_2(l) = l$, $(\rho_1 \cup \rho_2)(l) = l$ otherwise.

Product. The formation 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. Two iolss $L_1 = (I_1, O_1, T_1)$ and $L_2 = (I_2, O_2, T_2)$ 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 $T_2 \cap (I_1 \cup O_1 \cup T_1) = \emptyset$. The *shared* labels of composable iolss L_1 and L_2, written $L_1 \bowtie L_2$, are given by $(I_1 \cap O_2) \cup (O_1 \cap I_2)$. The *product* of two composable iolss L_1 and L_2 is the iol $L_1 \otimes L_2 = ((I_1 \cup I_2) \setminus (L_1 \bowtie L_2), (O_1 \cup O_2) \setminus (L_1 \bowtie L_2), T_1 \cup T_2 \cup (L_1 \bowtie L_2))$ which moves the shared labels to the internal labels. Two iotss A_1 and A_2 are *composable* if $L(A_1)$ and $L(A_2)$ are composable. 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

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

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 iotss is always binary. The product is commutative and associative. For a finite index set I, we write $\bigotimes_{i \in I} A_i$ for the product of the iotss A_i with $i \in I$.

3 Component Model with (A-)Synchronous Communication

We extend our component model presented in [3] to take into account not only synchronous but also asynchronous communication. By synchronous communication we understand a rendezvous mechanism where sender and receiver of a message 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. In our component model we distinguish between synchronous and asynchronous connectors which both are binary. For technical reasons we have considered in [3] also unary connectors, but apart from this point the component model described in the following is a conservative extension of the one in [3].

We consider components to be strongly encapsulated behaviours. Encapsulation is achieved by ports which regulate any interaction of components with their environment. Components can be hierarchically structured containing again an assembly of components and connectors. Figure 1 shows the metamodel of our component model. A *port*

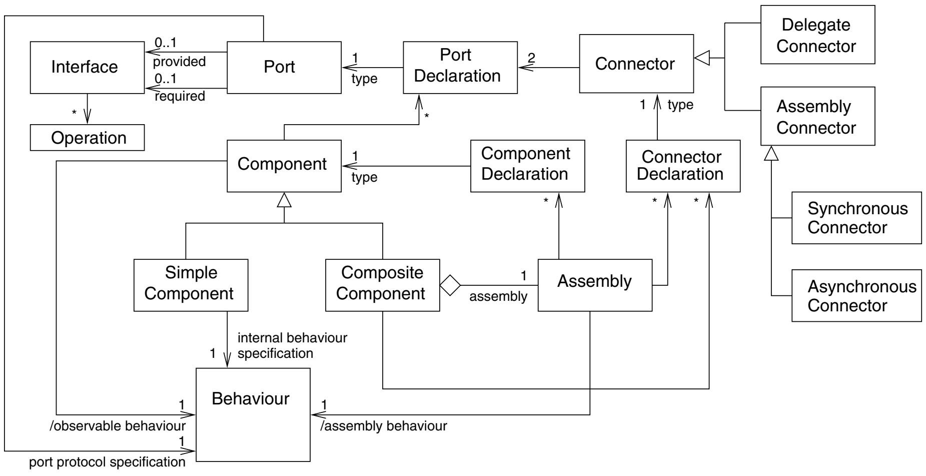


Fig. 1. Component metamodel

describes a view on a component. The operations offered by a port are summarised in its *provided interface*; the operations needed in its *required interface*. The sequencing of operation invocations issued and received by a port is described in a *port protocol specification*. To be precise, a port is in fact considered as a port type that can be used in local port declarations of a component.

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*, which introduce locally unique port names with corresponding port types, and an associated *observable behaviour* which describes the ordering of input and output actions of the component. In our metamodel a component represents in fact a component type that can be used in component declarations when building component assemblies. Each component should be correct with respect to its ports, i.e., the protocol of its ports should indeed be supported by the observable behaviour of the component. This correctness issue has been studied in [3]. For each simple component an *internal behaviour specification* is given which involves not only input and output actions but also transitions with internal actions. A composite component encapsulates an assembly of components. An *assembly* defines the internal structure of the composite component in terms of a set of local component declarations and local (binary) *assembly connector declarations* that connect local components via their ports. Assembly connectors can be synchronous or asynchronous. In a composite component, non-connected (open) ports of local components may be connected to so-called *relay ports* of the composite component, using *delegate connector declarations*. Also an assembly has an associated behaviour. As indicated by the slash symbol in Fig. 1 the observable behaviour of a component as well as the assembly behaviour are derived behaviours. The observable behaviour of simple components is derived from the components' given internal behaviour specification; for composite components the observable behaviour is derived from the behaviour of its assembly which in turn is derived from the observable

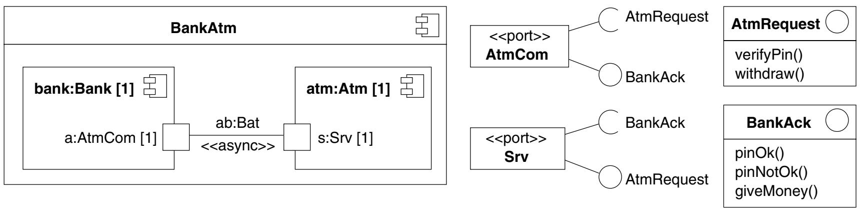


Fig. 2. Static structure of a simple Bank–Atm application

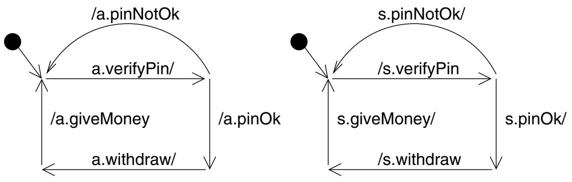


Fig. 3. Observable behaviour of Bank (left) and Atm (right)

behaviours of the local components within the assembly and their connections. In this paper we will particularly focus on assembly behaviours which depends on the synchronisation mechanisms used for the connectors.

Example 1 (Static structure and behaviours). Consider the simple Bank–Atm application in Fig. 2. A composite component type BankAtm contains an assembly of two simple components with types Bank and Atm introduced by the component declarations bank : Bank and atm : Atm.[1] The simple component types Bank and Atm have port declarations a : AtmCom and s : Srv resp. which are connected with an asynchronous assembly connector with name ab and type Bat.[2] The two simple component declarations and the connector form an assembly. The provided and required interfaces of the port types with their operations are depicted with the UML ball-and-socket notation on the right-hand side of Fig. 2. We do not consider operations with parameters here. If two ports are connected by an assembly connector, the provided interface of the one port has to be equal to the required interface of the other, and vice versa.[3]

Concerning behaviours we do not show port protocols and internal behaviour specifications of the simple component types Bank and Atm, but only give their derived observable behaviours in Fig. 3. Input and output messages are indicated by $p.m$/ and /$p.m$, respectively, where p is the port name on which the message is sent or received. Figure 4 shows the assembly behaviour of the asynchronously communicating Bank

[1] The UML2 declarations in Fig. 2 also show multiplicities, indicating how many instances of a component or port may exist. However, we only specified singletons (multiplicity 1) leaving the discussion of arbitrary multiplicities to future work.

[2] UML2 would allow for arbitrary n-ary connectors with $n > 2$ which we do not consider here.

[3] In general, one could use a more flexible condition such 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.

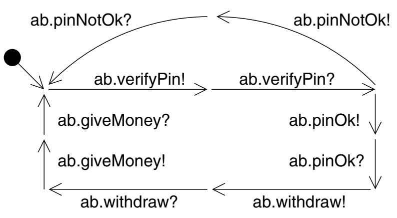


Fig. 4. Assembly behaviour of the Bank–Atm assembly

and Atm components. In this case a buffering behaviour of the connector ab is involved. Labels of the form ab.m! represent the action of sending a message m on the connector ab which, at the same time, will be put into the input buffer on the opposite side of the connector. Labels of the form ab.m? represent the action of taking a message out of the buffer. Taking out a message of the buffer semantically corresponds to the input of a message at a port as indicated in the observable behaviour of a component.

4 Formalisation of the Component Model

We will now provide a precise formalisation of our concepts for the static structure and for the dynamic behaviour of components with synchronous and asynchronous communication. For this purpose we complement the metamodel presentation of our component model with an algebraic description, which defines formally all previously mentioned concepts and behaviours in terms of algebraic structures and iotss resp.; see Sect. 2. In particular, we distinguish between those behaviours which have to be provided by the component developer and those that are computed (derived), by rendering the latter as definitions. We use *italics* to denote all kinds of derived operators. This section extends our formalisation in [3] to the asynchronous case.

4.1 Technical Prerequisites

Buffered I/O-Transition Systems. For asynchronous communication we need a mechanism for buffering (queueing) of messages. Technically, we use for this purpose buffered I/O-transition systems which model queues over a given set M of messages such that enqueueing a message m is an input action of the form $m!$ and dequeueing a message m 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 front is denoted by $m \cdot s$, extension of the back end by $s \cdot m$.

Definition 1 (Queue iots). *Let M be a set. The* queue iots *over M is given by $Q^{\lhd}_M = ((I, O, T), S, s_0, \Delta)$, where $I = \{m! \mid m \in M\}$, $O = \{m? \mid m \in M\}$, $T = \emptyset$; $S = \{s \mid s \in M^*\}$, $s_0 = \epsilon$; and $\Delta \subseteq S \times (I \cup O \cup T \cup \{\tau\}) \times S$ is the smallest relation such that*

1. *for all* $m! \in I$ *and all* $s \in S$*, there exists* $(s, m!, s \cdot m) \in \Delta$*,*
2. *for all* $m? \in O$ *and all* $m \cdot s \in S$*, there exists* $(m \cdot s, m?, s) \in \Delta$*.*

For an iol (I, O, T) and a set of labels M we define the sets of labels $I_{M?} = \{l? \mid l \in I \cap M\} \cup (I \setminus M)$, $O_{M!} = \{l! \mid l \in O \cap M\} \cup (O \setminus M)$ and analogously for $I_{M!}$ and $O_{M?}$. If $I \subseteq M$ or $O \subseteq M$ we write $I?$, $I!$ or $O!$, $O?$ respectively. The *relabelling for buffered communication* $\beta_M : (I, O, T) \to (I_{M?}, O_{M!}, T)$ is defined by $\beta_M(l) = l?$ if $l \in I \cap M$, $\beta_M(l) = l!$ if $l \in O \cap M$, and $\beta_M(l) = l$ otherwise.

Specialised Relabellings. For the computation of the behaviours of components and assemblies we employ several relabelling functions, which specialise the relabelling introduced in Sect. 2.1. 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 queue labels. We assume a primitive domain Nm of *names*.

A prefix relabelling prefixes all labels in an iots by a given name. 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 $A\rho_n$.

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.

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 $\sigma_{(X,y)} \circ \beta_M$ of a relabelling β_M for buffered communication (cf. above) with $M = \{x.l \in I \cup O \mid x \in X, l \in \mathrm{Nm}\}$ and a synchronisation relabelling $\sigma_{(X,y)}$. Finally, a *relay relabelling* $\rho_{(x,y)}$ is given by a match relabelling $\mu_{(X,y)}$ with $X = \{x\}$.

4.2 Formalisation of Ports and Connectors

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

If ports are used for asynchronously communicating components, a queue iots is defined with respect to the messages provided at the particular port.

Definition 2 (Queue of a port). *The* queue *of a port (type) P is given by $que(P) = Q^{\lhd}_{\mathrm{msg}(\mathrm{prv}(P))}$.*

For each port $P \in \mathrm{Port}$ we assume given its *protocol (specification)*, written $\mathrm{prot}(P)$, which is an I/O-transition system $((I, O, T), Q, q_0, \Delta)$ with $I = \mathrm{msg}(\mathrm{prv}(P))$, $O = \mathrm{msg}(\mathrm{req}(P))$ and $T = \emptyset$.

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

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 $\mathrm{req}(P_1)$ has to be equal to the provided interface $\mathrm{prv}(P_2)$ and vice versa. There are again two disjoint sub-domains $\mathrm{AsynchConn} \subseteq \mathrm{AsmConn}$, $\mathrm{SynchConn} \subseteq \mathrm{Conn}$ 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. Hence, they must show a buffering behaviour on each end of the connector in accordance with the messages that can be received (i.e. are provided) at a particular port.

Definition 3 (Buffering connector behaviour). *The* buffering behaviour *of an asynchronous connector $k : (p : P, q : Q)$ is given by $buf(k : (p : P, q : Q)) = k.(que(P) \otimes que(Q))$.*

To obtain a uniform definition of assembly behaviour below (Def. 4) we need for technical reasons a notion of "empty iots" which acts as a neutral element w.r.t. the product of iotss. Define $\mathbf{1}$ to be an iots (L, S, s_0, Δ) with $\bigcup L = \emptyset$, $S = \{s_0\}$ and $\Delta = \emptyset$. If a connector $k : K$ is synchronous we set $buf(k : K) = \mathbf{1}$.

4.3 Formalisation of Components and Assemblies

Components. We assume a domain Cmp of components (more precisely, component types) and a function $\mathrm{ports} : \mathrm{Cmp} \to \wp\mathrm{PortDcl}$ returning the ports declared for a component. For a component C and port declaration $p : P$ we write $C[p : P]$ to indicate that $p : P \in \mathrm{ports}(C)$. 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 $\mathrm{nm} : \mathrm{CmpDcl} \to \mathrm{Nm}$ for the name and a function $\mathrm{ty} : \mathrm{CmpDcl} \to \mathrm{Cmp}$ for the component (type); we write $c : C$ for a component declaration d with $\mathrm{nm}(d) = c$ and $\mathrm{ty}(d) = C$. The ports of a component declaration are given by $\mathit{ports}(c : C) = \{c.p : P \mid p : P \in \mathrm{ports}(C)\}$.

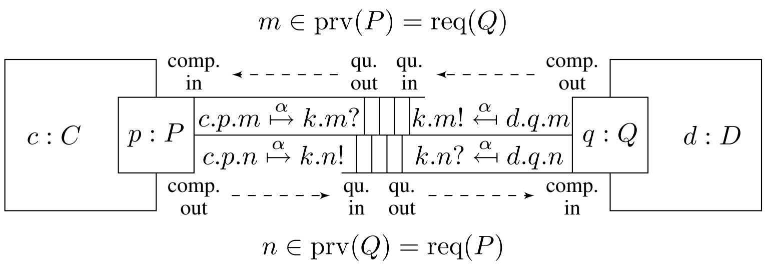
For each component (type) $C \in \mathrm{Cmp}$ there is a derived *observable behaviour*, written $\mathit{obs}(C)$, which is an iots $((I, O, T), Q, q_0, \Delta)$ with $I = \bigcup\{p.\mathrm{msg}(\mathrm{prv}(P)) \mid p : P \in \mathrm{ports}(C)\}$, $O = \bigcup\{p.\mathrm{msg}(\mathrm{req}(P)) \mid p : P \in \mathrm{ports}(C)\}$ and $T = \emptyset$. Hence, the observable labels of a component (type) 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. The only additional action that can occur in the observable behaviour of a component is the invisible action τ. As already indicated in the component metamodel in Fig. 1 the observable behaviour of a component is a derived behaviour. Its definition depends on whether the component is simple or composite; cf. Def. 5 and Def. 6 below. The *observable behaviour* of a component declaration $c : C$ is given by $\mathit{obs}(c : C) = c.\mathit{obs}(C)$.

Assemblies. Let us now formalise the static structure and the behaviour of component assemblies. 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 $\mathrm{cmps} : \mathrm{Asm} \to \wp\mathrm{CmpDcl}$ returning an assembly's declared components and $\mathrm{conns} : \mathrm{Asm} \to \wp\mathrm{ConnDcl}$ 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 $\mathit{acs}(a) \subseteq \mathrm{conns}(a)$ such that $k : K \in \mathit{acs}(a)$ iff $K \in \mathrm{AsynchConn}$, and the subset of synchronous connectors by $\mathit{scs}(a) \subseteq \mathrm{conns}(a)$ such that $k : K \in \mathit{scs}(a)$ iff $K \in \mathrm{SynchConn}$.

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

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

Let us now 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 observable 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. 3) plays a crucial role. Moreover, some matching relabellings are necessary to achieve the desired behaviour.

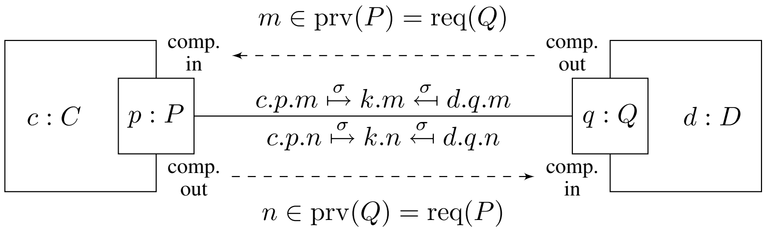


$m \in \mathrm{prv}(P) = \mathrm{req}(Q)$
comp. in
qu. out
qu. in
comp. out
$c : C$
$p : P$
$c.p.m \overset{\alpha}{\mapsto} k.m?$
$c.p.n \overset{\alpha}{\mapsto} k.n!$
$k.m! \overset{\alpha}{\hookleftarrow} d.q.m$
$k.n? \overset{\alpha}{\hookleftarrow} d.q.n$
$q : Q$
$d : D$
comp. out
qu. in
qu. out
comp. in
$n \in \mathrm{prv}(Q) = \mathrm{req}(P)$

Fig. 5. Assembly with asynchronous connector

Figure 5 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 $que(Q)$ (cf. Def. 2) which allows inputs of the form $k.n!$ with n being a message according to the provided interface of Q. To achieve that the issued message $c.p.n$ will indeed be put into the queue $que(Q)$, we use a matching relabelling α which maps $c.p.n$ to $k.n!$. The message n can be dequeued from $que(Q)$ later on with the action $k.n?$. Since the component d inputs on its port q messages of the form $d.q.n$ we use again the matching relabelling α which now maps $d.q.n$ to $k.n?$. The communication in the other direction works analogously.

Figure 6 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.



$m \in \mathrm{prv}(P) = \mathrm{req}(Q)$
comp. in
comp. out
$c : C$
$p : P$
$c.p.m \overset{\sigma}{\mapsto} k.m \overset{\sigma}{\hookleftarrow} d.q.m$
$c.p.n \overset{\sigma}{\mapsto} k.n \overset{\sigma}{\hookleftarrow} d.q.n$
$q : Q$
$d : D$
comp. out
comp. in
$n \in \mathrm{prv}(Q) = \mathrm{req}(P)$

Fig. 6. Assembly with synchronous connector

For general assemblies, we apply the match relabellings defined in Sect. 4.1 to define the relabelling α for asynchronous and σ for synchronous connectors.

$$\alpha \equiv \bigcup\{\alpha_{(\{c.p,d.q\},k)} \mid k : K \in acs(a) \,.\, \mathrm{ports}(K) = \{c.p : P, d.q : Q\}\} \ ,$$
$$\sigma \equiv \bigcup\{\sigma_{(\{c.p,d.q\},k)} \mid k : K \in scs(a) \,.\, \mathrm{ports}(K) = \{c.p : P, d.q : Q\}\} \ .$$

We have now the technical ingredients to define the behaviour of an assembly. In the definition the successive application of α and σ cannot lead to conflicts because both relabellings are disjoint. Note also that in the case of synchronous connectors $buf(k : K)$ is trivial as explained in Sect. 4.2. 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 4 (Assembly behaviour). *The* behaviour *of an assembly a is given by*

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

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

Example 2 (Assembly behaviour). The static structure of the Bank–ATM application in Fig. 2 is formally represented by an assembly $\langle\mathsf{bank} : \mathsf{Bank}, \mathsf{atm} : \mathsf{Atm}; \mathsf{ab} : \mathsf{Bat}\rangle$. The assembly behaviour, shown in Fig. 4, is obtained from the composition of the observable behaviours of the components Bank and Atm with the buffering behaviour of the asynchronous connector Bat:

$$beh(\langle\mathsf{bank} : \mathsf{Bank}, \mathsf{atm} : \mathsf{Atm}; \mathsf{ab} : \mathsf{Bat}\rangle) =$$
$$obs(\mathsf{bank} : \mathsf{Bank})\alpha\sigma \otimes obs(\mathsf{atm} : \mathsf{Atm})\alpha\sigma \otimes buf(\mathsf{ab} : \mathsf{Bat}) \ .$$

Simple Components. We assume a sub-domain $\mathrm{SCmp} \subseteq \mathrm{Cmp}$ of simple components. Each $SC \in \mathrm{SCmp}$ has a user defined *internal behaviour specification* $\mathrm{beh}(SC)$, which is an iots $((I, O, T), Q, q_0, \Delta)$ with $I = \{p.\mathrm{msg}(\mathrm{prv}(P)) \mid p : P \in \mathrm{ports}(SC)\}$ and $O = \{p.\mathrm{msg}(\mathrm{req}(P)) \mid p : P \in \mathrm{ports}(SC)\}$. The observable behaviour of a simple component SC is derived from its internal behaviour specification by hiding all internal labels. Technically this is achieved with the hiding operator ξ; see Sect. 2.

Definition 5 (Observable behaviour of simple component). *The* observable behaviour *of a simple component SC is given by $obs(SC) = \mathrm{beh}(SC)\xi$.*

Example 3 (Observable behaviours). The observable behaviour hides internal transitions, i.e. relabels internal transition to τ. In order to keep our running example simple and illustrative we refrained from modelling internal behaviour and assumed observable behaviours without τ transitions instead (cf. Fig. 3).

Composite Components. 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 $\mathrm{asm} : \mathrm{CCmp} \to \mathrm{Asm}$ returning the underlying assembly of a composite component, and $\mathrm{conns} : \mathrm{CCmp} \to \wp\mathrm{ConnDcl}$ 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 $k : K \in \mathrm{conns}(CC)$, then $K \in \mathrm{DlgConn}$; (ii) all open ports of the $\mathrm{asm}(CC)$ are connected, i.e., for all $c.p : P \in \mathit{open}(\mathrm{asm}(CC))$ there is $k : K \in \mathrm{conns}(CC)$ such that $c.p : P \in \mathrm{ports}(K)$; and (iii) all relay ports are connected, i.e., for all $r : R \in \mathrm{ports}(CC)$ there is a unique $k : K \in \mathrm{conns}(CC)$ with $\mathrm{ports}(K) = \{c.p : P, r : R\}$ and $c.p : P \in \mathit{open}(\mathrm{asm}(CC))$.

The observable behaviour of a composite component is derived from the behaviour of its underlying assembly by hiding all internal actions, which in the case of assemblies are the communications on the connectors, and 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 6 (Observable behaviour of composite component). *The* observable behaviour *of a composite component CC is given by*

$$\mathit{obs}(CC) = (\mathit{beh}(\mathrm{asm}(CC))\xi)\rho\ ,$$

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

We write $\langle a; \mathcal{P}; \mathcal{K}\rangle$ for a composite component CC with assembly $\mathrm{asm}(CC) = a$, set of (relay) port declarations $\mathrm{ports}(CC) = \mathcal{P}$ and set of (delegate) connector declarations $\mathrm{conns}(CC) = \mathcal{K}$.

Example 4 (Observable behaviour of composite components). Since the Bank–Atm application is a closed system, the observable behaviour of the composite component $\langle\langle \mathsf{bank} : \mathsf{Bank}, \mathsf{atm} : \mathsf{Atm}; \mathsf{ab} : \mathsf{Bat}\rangle; \emptyset; \emptyset\rangle$ consists of τ-transitions only.

4.4 Buffered Components

The assembly behaviour has been defined on the basis of connectors which may show an asynchronous buffering behaviour. We show that the assembly behaviour can also be computed by rearranging the buffers in such a way that they do not belong to the connectors but to the components. For this purpose we introduce a new kind of component behaviour which integrates observable behaviour and message buffers for a single component based on a notion of buffered iotss.

An iots $A = (L, S, s_0, \Delta)$ is *without queue labels* if L does not contain labels of the form $m?$ or $m!$ (dequeue and enqueue labels of queue iotss; cf. Def.1).

Lemma 1. *If $A = ((I, O, T), S, s_0, \Delta)$ is without queue labels, $X \subseteq I$ and $Y \subseteq O$, then $A\beta_{X\cup Y}$ and $Q^{\lhd}_X$ are composable.* □

The relabelling $\beta_{X\cup Y}$, defined in Sect. 4.1, prepares A on the one hand for the synchronisation with its queue $Q^{\lhd}_X$, and, on the other hand for synchronisation with an iots which provides matching inputs for the asynchronous outputs of A on Y.

Definition 7 (Buffered iots). *Let $A = ((I, O, T), S, s_0, \Delta)$ be an iots without queue labels. Let $X = \{X_1, \dots, X_n\}$ where $X_i \subseteq I$ and $X_i \cap X_j = \emptyset$ for all $i \neq j \in \{1, \dots, n\}$. The* buffered iots for *A with buffered input X and buffered output $Y \subseteq O$ is given by*

$$\Omega_{X,Y}(A) = A\beta_{X_1 \cup \dots \cup X_n \cup Y} \otimes Q^{\lhd}_{X_1} \otimes \dots \otimes Q^{\lhd}_{X_n} ,$$

If $X = I$ and $Y = O$, then the iots is completely buffered *and we write $\Omega(A)$.*

Lemma 2. *If $A = ((I, O, T), S, s_0, \Delta)$ is without queue labels, $X = \{X_1, \dots, X_n\}$ where $X_i \subseteq I$ and $X_i \cap X_j = \emptyset$ for all $i \neq j \in \{1, \dots, n\}$, and $Y \subseteq O$, then $L(\Omega_{X,Y}(A)) = (I_{(X_1 \cup \dots \cup X_n)!}, O_{Y!}, T \cup (I_{(X_1 \cup \dots \cup X_n)?} \setminus I))$.* □

Buffered iotss are later used to develop and analyse notions of refinement and compatibility on the level of iotss. The results are then applied to our formal component model. Therefore, we detail in the following on buffered iotss and their intended application as a formal representation of asynchronously communicating components.

Example 5 (Buffered iots). Buffered iotss for the observable behaviour of components are obtained from their composition with input queues defined w.r.t. the ports of the given component. For instance, the Bank component is equipped with one port only, therefore we have $\Omega_{X,Y}(obs(\mathsf{Bank})) = obs(\mathsf{Bank})\beta_{X \cup Y} \otimes Q^{\lhd}_X$, where $X = \{\{\mathsf{a}.m \mid m \in \{\mathsf{verifyPin}, \mathsf{withdraw}\}\}\}$ and $Y = \{\mathsf{a}.m \mid m \in \{\mathsf{pinOk}, \mathsf{pinNotOk}, \mathsf{giveMoney}\}\}$.

Definition 8 (Communication behaviour of component). *The* communication behaviour *of a component C buffered on a set of port declarations $\mathcal{P}$ is given by*

$$com_{\mathcal{P}}(C) = \Omega_{X,Y}(obs(C)) ,$$

where $X = \{X_{p:P} \mid p : P \in \mathcal{P} \cap \mathrm{ports}(C)\}$, $X_{p:P} = \{p.m \mid m \in \mathrm{msg}(\mathrm{prv}(P))\}$, $Y = \{p.m \mid p : P \in \mathcal{P} \cap \mathrm{ports}(C) \wedge m \in \mathrm{msg}(\mathrm{req}(P))\}$.

The communication behaviour of a component declaration $c : C$ w.r.t. a set of port declarations $\mathcal{P} \subseteq \mathrm{ports}(C)$ is given by $com_{\mathcal{P}}(c : C) = c.com_{\mathcal{P}}(C)$.

In order to obtain a characterisation of assembly behaviour in terms of communicating buffered components we have to ensure commutativity and associativity of compositions of buffered iotss. Concerning composability we record only the special case of completely buffered iotss, which is later needed in our analysis.

Lemma 3. *Let A and B be iotss without queue labels. If A and B are composable, then $\Omega(A)$ and $\Omega(B)$ are composable.* □

The lemma holds also for arbitrary buffered iotss, if we ensure that the input partitions determining the asynchronous input of one iots is consistent with the relabelling for asynchronous output of its communication partner. As a consequence the composition of buffered iotss is associative and commutative. The proof of these facts is tedious but rather straightforward.

Definition 8 includes the synchronisation of the observable behaviour of a component with its input queues. By composition of communication behaviours, we synchronise output transitions of one component behaviour with input transitions of the queues of other components. Hence we can obtain an assembly behaviour by composition of communication behaviours of components which is equivalent to Def. 4. For taking into account the names of asynchronous connectors, we replace the asynchronous relabelling α used in Def. 4 by a slightly simpler relabelling, defined by $\kappa \equiv \bigcup\{\mu_{(\{c.p,d.q\},k)} \mid k : K \in acs(a) \,.\, \mathrm{ports}(K) = \{c.p : P, d.q : Q\}\}$.

Proposition 1. *If a is an assembly, then $beh(a) = \bigotimes_{c:C \in \mathrm{cmps}(a)}(com_{\mathcal{P}}(c : C)\kappa\sigma)$, where $\mathcal{P} = \bigcup\{\mathrm{ports}(K) \mid k : K \in acs(a)\}$.* □

5 Connection-Safe Assemblies

A safe communication of two components over a synchronous connector is characterised by the fact that if one component is about to send a message the other component is indeed willing to accept this message. For the iotss underlying the components this means that if the one iots has reached a state where it does an output, the other iots is in a state where it does the corresponding input. This idea of synchronous safe communication is captured by the following notion of compatibility of iotss as introduced by Gouda, Manning, and Yu for communicating finite state machines [17] and used by de Alfaro and Henzinger for iotss [4], which is based on the reachable states of iotss: 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 $a \in \bigcup L \cup \{\tau\}$ and an $s' \in S$ with $(s, a, s') \in \Delta$, then $s' \in \mathscr{R}(A)$.

Definition 9 (Compatibility). *Let $A = ((I_A, O_A, T_A), S_A, s_{0,A}, \Delta_A)$ and $B = ((I_B, O_B, T_B), S_B, s_{0,B}, \Delta_B)$ be composable iotss. B is a* compatible context *for A, if for all $l \in O_A \cap I_B$ and all $(s_A, s_B) \in \mathscr{R}(A \otimes B)$, if $(s_A, l, s'_A) \in \Delta_A$, then there exists $(s_B, l, s'_B) \in \Delta_B$. The iotss A and B are* compatible *if A is a compatible context for B and vice versa.*

Example 6 (Compatible iotss). The iotss representing the observable behaviours of the components Bank and Atm in Fig. 3 are obviously compatible. All outputs are immediately synchronised in both directions (modulo port relabelling).

For asynchronously communicating components which are connected by buffers the situation of safe communication is different: We have to ensure that each message sent out by one component is eventually understood by the receiving component. For technical reasons we restrict our attention to infinite communication sequences. Then safe communication means that if we observe an infinite communication sequence with output labels putting a message into a buffer and input labels taking a message from a buffer we have to be able to pair off the corresponding output and input labels. We thus base the notion of buffered compatibility of buffered iotss as an analogue to (synchronous) compatibility of iotss on infinite label sequences and require for buffered compatibility a pairing function for sending and taking. An *infinite run* of an iots $A = (L, S, s_0, \Delta)$ is an infinite sequence $s_0, l_0, s_1, l_1, \ldots$ with $s_n \in S$, $l_n \in \bigcup L$, and $(s_n, l_n, s_{n+1}) \in \Delta$ for all $n \in \mathbb{N}$. An *infinite weak trace* of A is a sequence $l_0, l_1, \ldots$ with $l_n \in \bigcup L$ such that there is a run $s_0, l_0, s_1, l_1, \ldots$ of A.

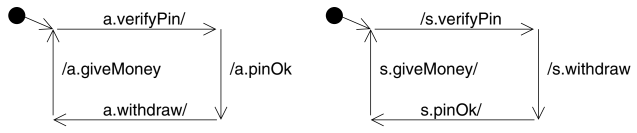


Fig. 7. Example for buffered compatible iotss

Definition 10 (Buffered compatibility). *Let (L, S, s_0, Δ) be an iots and $\mu : U \to V$ a mapping with $U, V \subseteq \bigcup L$. An infinite weak trace $l_0, l_1, \ldots$ of (L, S, s_0, Δ) is μ-*buffered compatible, *if for each $u \in U$ there is a bijection $\varphi_u : \{k \in \mathbb{N} \mid l_k = u\} \to \{k \in \mathbb{N} \mid l_k = \mu(u)\}$ with $k < \varphi_u(k)$.*

Let A and B be composable iotss without queue labels and $L(A) = (I_A, O_A, T_A)$ and $L(B) = (I_B, O_B, T_B)$. Let $\mu : \{m! \mid m \in O_A \cup O_B\} \to \{m? \mid m \in I_A \cup I_B\}$ be defined by $\mu(m!) = m?$. An infinite weak trace of $\Omega(A) \otimes \Omega(B)$ is buffered compatible, *if it is μ-buffered compatible. A and B are* buffered compatible, *if all infinite weak traces of $\Omega(A) \otimes \Omega(B)$ are buffered compatible.*

Example 7 (Buffered compatible iotss). Asynchronous communication allows for simultaneous sending of messages as illustrated, for instance, in the iotss of Fig. 7. Compared to the behaviours known from Fig. 3, the order of s.withdraw and s.pinOk in the right-hand iots was swapped and both iots were reduced to one path. The composition of the corresponding buffered iotss results in an iots where the possibility of simultaneous sending is modelled by the respective queue actions (cf. Fig. 8).

Obviously the iotss in Fig. 7 are not synchronously compatible, due to the output of the messages a.pinOk and s.withdraw. However, they are buffered compatible. Figure 8 shows the product iots, which would be obtained along appropriate relabellings (cf. Prop. 1) and an asynchronous connector ab : Bat as above. The infinite weak traces of the product allow to match the enqueue actions ab.verifyPin!, ab.withdraw!, etc. with their

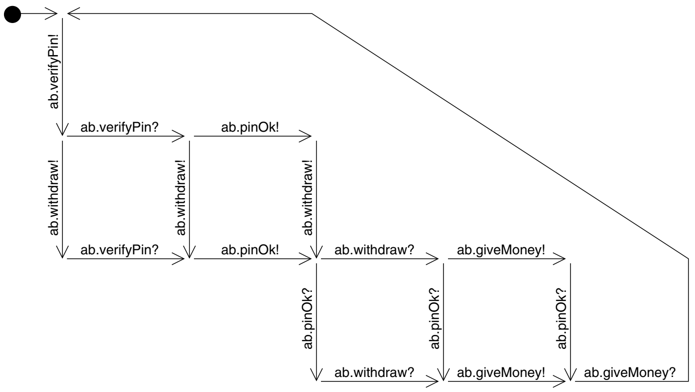


Fig. 8. Product of buffered iotss with simultaneous sending

dequeuing counterparts ab.verifyPin?, ab.withdraw?, etc. as required by the definition of μ-buffered compatibility.

In the context of component assemblies, compatibility of two synchronously or asynchronously communicating iotss has to be lifted to an arbitrary number of communicating components. We therefore introduce the notion of connection-safe assemblies.

Definition 11 (Connection-safety). *An assembly* $a = \langle c_1 : C_1, \ldots, c_n : C_n; \mathcal{K}\rangle$ *is* connection-safe*, if for all connector declarations* $k : (c_i.p : P, c_j.q : Q) \in \mathcal{K}$ *the following conditions hold:*

1. *If* k *is synchronous, let obs*$(c_i : C_i) = (L_{c_i}, S_{c_i}, s_{0,c_i}, \Delta_{c_i})$ *and obs*$(c_j : C_j) = (L_{c_j}, S_{c_j}, s_{0,c_j}, \Delta_{c_j})$*. Then for all* $(s_{c_1}, \ldots, s_{c_n}, K) \in \mathscr{R}(\mathrm{beh}(a))$*, if* $(s_{c_i}, c_i.p.m, s'_{c_i}) \in \Delta_{c_i}$ *for some* $m \in \mathrm{msg}(\mathrm{req}(P))$*, then there is a* $(s_{c_j}, c_j.q.m, s'_{c_j}) \in \Delta_{c_j}$*; and if* $(s_{c_j}, c_j.q.m, s'_{c_j}) \in \Delta_{c_j}$ *for some* $m \in \mathrm{msg}(\mathrm{req}(Q))$*, then there is a* $(s_{c_i}, c_i.p.m, s'_{c_i}) \in \Delta_{c_i}$*.*
2. *If* k *is asynchronous, let* $\mu : \{k.m! \mid m \in \mathrm{msg}(\mathrm{req}(P)) \cup \mathrm{msg}(\mathrm{req}(Q))\} \to \{k.m? \mid m \in \mathrm{msg}(\mathrm{prv}(Q)) \cup \mathrm{msg}(\mathrm{prv}(P))\}$ *with* $\mu(k.m!) = k.m?$*. Then all infinite weak traces of* $\mathrm{beh}(a)$ *are* μ*-buffered compatible.*

Note that for assemblies consisting of just two components with one port each and being connected by a single either synchronous or asynchronous connector, connection safety just means (synchronous) compatibility or buffered compatibility of the iotss underlying the communication behaviour of components of Def. 8.

The different concepts of compatibility for rendezvous and buffered communication raise the question whether synchronous compatibility of two iotss induces their buffered compatibility when they are put into an asynchronous context; we concentrate on closed compositions, where an iots $((I, O, T), S, s_0, \Delta)$ is *closed* if $I = O = \emptyset$. In order to answer this question, we first extend a result by Cécé and Finkel [15, Thm. 35] that, under some restrictions, two compatible finite state machines yield, when communicating through queues, a so-called half-duplex system. In our context of iotss, a composition $\Omega(A) \otimes \Omega(B)$ is *half-duplex*, if in every reachable state $((s_A, q_A), (s_B, q_B)) \in \mathscr{R}(\Omega(A) \otimes \Omega(B))$ one of the queues is empty: $q_A = \epsilon \vee q_B = \epsilon$. The proof of this result was by contradiction and used the restriction that the two compatible communicating finite state machines are deterministic and have no so-called mixed states, i.e., states where both an input and an output can happen. For iotss, this corresponds to input separation, that is, for each state showing some outgoing input transition all outgoing transitions are labelled by inputs: An iots $A = (L, S, s_0, \Delta)$ with $L = (I, O, T)$ is *input separated*, if for all $s \in \mathscr{R}(A)$ with $(s, l, s') \in \Delta$ for some $s' \in S$ and $l \in I$, then $\{a \in \bigcup L \cup \{\tau\} \mid \exists s' \in S \,.\, (s, a, s') \in \Delta\} \subseteq I$. For example the iotss in Fig. 3 are input separated, since all states with outgoing input transitions show only input transitions. Input separation may be understood as a property which reflects single-threaded execution: output and internal transitions succeeding an input are considered to encode the reaction of the component to this input.

Moreover, the restriction to deterministic iotss would not be appropriate for our setting because in the context of invisible actions non-determinism arises quite naturally. After establishing the result that compatible, input separated iots A and B induce a half-duplex asynchronous system $\Omega(A) \otimes \Omega(B)$ by a direct proof based on an invariant, we show that if we additionally require A and B to always eventually take an input, then A and B are buffered compatible. An iots $A = (L, S, s_0, \Delta)$ with $L = (I, O, T)$ is *always eventually inputting*, if all weak infinite traces $l_0, l_1, \dots$ show infinitely many $l_n \in I$.

Lemma 4. *Let A and B be composable, input separated iotss without queue labels, and let $A \otimes B$ be closed. If A and B are compatible, then $\Omega(A) \otimes \Omega(B)$ is half-duplex.*

Proof. Let us first fix some terminology: For an iots (L, S, s_0, Δ) the *transitive closure* of Δ is the relation $\Delta^* \subseteq S \times \bigcup L^* \times S$ defined inductively as follows: $(s, \epsilon, s') \in \Delta^*$, if $s = s'$; $(s, l \cdot \lambda, s') \in \Delta^*$, if there is an $s'' \in S$ with $(s, l, s'') \in \Delta$, and $(s'', \lambda, s') \in \Delta^*$. The *transitive τ-closure* of Δ is the relation $\hat{\Delta}^* \subseteq S \times \bigcup L^* \times S$ defined inductively by: $(s, \epsilon, s') \in \hat{\Delta}^*$, if $(s, \tau, s') \in \hat{\Delta}$; $(s, l \cdot \lambda, s') \in \hat{\Delta}^*$, if there is an $s'' \in S$ with $(s, l, s'') \in \hat{\Delta}$, and $(s'', \lambda, s') \in \hat{\Delta}^*$. The *safe label sequences* $\Delta^*(s) \subseteq \bigcup L^*$ in a state $s \in S$ are inductively given by: $\epsilon \in \Delta^*(s)$ for all $s \in S$; $l \cdot \lambda \in \Delta^*(s)$, if there is an $s' \in S$ with $(s, l, s') \in \Delta$, and for all $s' \in S$ with $(s, l, s') \in \Delta$ it holds that $\lambda \in \Delta^*(s')$. For an iots $A = ((I, O, T), S, s_0, \Delta)$ we write I_A for I, O_A for O and similarly for the other parts.

Define, using the hiding operator ξ defined in Sect. 2.1,

$$
\begin{aligned}
R = \{(((s_A, q_A), (s_B, q_B)), (r_A, r_B)) \mid \\
&((s_A, q_A), (s_B, q_B)) \in \mathscr{R}(\Omega(A) \otimes \Omega(B)) \wedge (r_A, r_B) \in \mathscr{R}(A \otimes B) \wedge \\
&((q_A = \epsilon \wedge q_B = \epsilon \wedge (s_A, s_B) = (r_A, r_B)) \vee \\
&\ (q_A = \epsilon \wedge q_B \neq \epsilon \wedge s_B = r_B \wedge (r_A, q_B, s_A) \in \hat{\Delta}^*_{A\xi} \wedge q_B \in \Delta^*_B(r_B) \vee \\
&\ (q_A \neq \epsilon \wedge q_B = \epsilon \wedge s_A = r_A \wedge (r_B, q_A, s_B) \in \hat{\Delta}^*_{B\xi} \wedge q_A \in \Delta^*_A(r_A)))\}
\end{aligned}
$$

We show that for all reachable $((s_A, q_A), (s_B, q_B)) \in \mathscr{R}(\Omega(A) \otimes \Omega(B))$ it holds that $\exists (r_A, r_B) \,.\, (((s_A, q_A), (s_B, q_B)), (r_A, r_B)) \in R$. Then, by definition of R, always one of the queues in $\Omega(A) \otimes \Omega(B)$ is empty. In fact, $(((s_{0,A}, \epsilon), (s_{0,B}, \epsilon)), (s_{0,A}, (s_{0,B}))) \in R$. Let $(((s_A, q_A), (s_B, q_B)), a, ((s'_A, q'_A), (s'_B, q'_B))) \in \Delta_{\Omega(A) \otimes \Omega(B)}$. We only consider transitions originating from $\Omega(A)$, the cases for transitions from $\Omega(B)$ are symmetric.

If $a \in T_A \cup \{\tau\}$, then $(s_A, a, s'_A) \in \Delta_A$ and $q_A = q'_A$, $(s_B, q_B) = (s'_B, q'_B)$. If $q_A = q_B = \epsilon$, then $(s_A, s_B) = (r_A, r_B)$ and thus $(((s'_A, q_A), (s_B, q_B)), (s'_A, r_B)) \in R$. If $q_A = \epsilon$ and $q_B \neq \epsilon$, then $(((s'_A, q_A), (s_B, q_B)), (r_A, r_B)) \in R$. But $q_A \neq \epsilon$ and $q_B = \epsilon$ is impossible, as then $s_A = r_A$ and hence A would not be input separated, since $q_A \in \Delta^*_A(r_A)$.

If $a \in T_{\Omega(A)} \setminus T_A$, then $a = m?$ for some $m \in I_A$, $(s_A, m, s'_A) \in \Delta_A$, $q_A = m \cdot q'_A$, $(s_B, q_B) = (s'_B, q'_B)$. Thus $s_A = r_A$, $(r_B, q_A, s_B) \in \hat{\Delta}^*_{B\xi}$, and $q_A \in \Delta_A(s_A)$. Moreover, $q'_A \in \Delta^*_A(s'_A)$. If $q'_A = \epsilon$, then $((r_A, r_B), m, (s'_A, s_B)) \in \Delta^*_{A\xi \otimes B\xi}$ and $(((s'_A, q'_A), (s_B, q_B)), s'_A, s_B) \in R$; if $q'_A \neq \epsilon$, $((r_A, r_B), m, (s'_A, r'_B)) \in \hat{\Delta}^*_{A\xi \otimes B\xi}$ with $(r'_B, q'_A, s_B) \in \hat{\Delta}_{B\xi}$ and hence $(((s'_A, q'_A), (s_B, q_B)), r'_A, r'_B) \in R$.

If $a \in O_{\Omega(A)}$, then $a = m!$ for some $m \in O_A$, $(s_A, m, s'_A) \in \Delta_A$, $q_A = q'_A$, $s_B = s'_B$, $q'_B = q_B \cdot m$. If $q_A = q_B = \epsilon$, then $(s_A, s_B) = (r_A, r_B)$. As $(r_A, r_B) \in \mathscr{R}(A \otimes B)$ and because A and B are compatible, there is a state $(s'_A, s'_B) \in S_{A \otimes B}$ with $((s_A, s_B), m, (s'_A, s'_B)) \in \Delta_{A \otimes B}$ and thus $(((s'_A, q_A), (s_B, q_B \cdot m)), (r_A, r_B)) \in R$. If $q_A = \epsilon$ and $q_B \neq \epsilon$, then $s_B = r_B$. We have $(r_A, q_B \cdot m, s'_A) \in \hat{\Delta}^*_{A\xi}$. In order to show that $q_B \cdot m \in \Delta^*_B(r_B)$, let $r'_B \in S_B$ with $(r_B, q_B, r'_B) \in \Delta^*_B$. Then $((r_A, r_B), q_B, (s_A, r'_B)) \in \hat{\Delta}_{A\xi \otimes B\xi}$ and, in particular, $(s_A, r'_B) \in \mathscr{R}(A \otimes B)$ and hence, by the compatibility of A and B, we have $(r'_B, m, r''_B) \in \Delta_B$. Thus $(((s'_A, q_A), (s_B, q_B \cdot m)), (r_A, r_B)) \in R$. But $q_A \neq \epsilon$ and $q_B = \epsilon$ is impossible, as then $s_A = r_A$ and hence A would not be input separated, since $q_A \in \Delta^*_A(r_A)$. □

Theorem 1. *Let A and B be composable, input separated iotss without queue labels, and let $A \otimes B$ be closed. Let A and B be always eventually inputting. If A and B are compatible, then A and B are buffered compatible.*

Proof. Let A and B be compatible. Then $\Omega(A) \otimes \Omega(B)$ is half-duplex by Lem. 4. Consider an infinite weak trace λ of $\Omega(A) \otimes \Omega(B)$. As A and B are always eventually inputting, λ shows infinitely many labels of the form $m?$ with m an input label of A and infinitely many labels of the form $n?$ with n an input label of B. In each state of $\Omega(A) \otimes \Omega(B)$ with an outgoing transition with a label marked with ? the corresponding queue of A and B resp. is not empty, thus the queue of the other iots is empty. Hence each output of $\Omega(A)$ and $\Omega(B)$ is eventually answered by an input of A and B resp. and hence λ is buffered compatible. □

Example 8 (Compatiblity and buffered compatibility). The theorem is applicable to the iotss given by Fig. 3. The iotss are obviously input separated. They are compatible by Ex. 6, and they are always eventually inputting, since all of their weak infinite traces show infinitely many input labels. Therefore the iotss are indeed buffered compatible. Note that the input assumption for A and B is necessary. The iots on the left-hand side of Fig. 9 is not always eventually inputting and even though the iotss are synchronously compatible, they are not buffered compatible. The sender may proceed infinitely often with output actions while the receiver never dequeues resulting in an infinite weak trace that is not μ-buffered compatible.

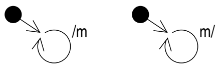


Fig. 9. Buffered incompatible I/O-transition systems

As witnessed by Ex. 7 (Fig. 7), the converse of Thm. 1, that buffered compatibility induces synchronous compatibility of the non-queued iotss, is not true in general.

6 Compositional Refinement of Connection-Safe Assemblies

When substituting a refined version of a component for another component in an assembly context, it should be ensured that relevant properties of the original assembly are preserved. In the following we introduce a notion of refinement of components based on alternating simulations of interface automata [4] and show that, at least for binary assemblies, component refinement is compositional and preserves connection-safety of assemblies.

6.1 Refinement of I/O-Transition Systems and Components

When an alternating simulation, as defined by de Alfaro and Henzinger [4], puts two states of an abstract and a concrete behaviour in relation, each input of the abstract behaviour must be accepted as an input of the concrete behaviour and, conversely, each output of a concrete behaviour must be an output of the abstract behaviour; these requirements correspond to clauses (1) and (2) in the definition below. We adopt this general idea, but formally extend the definition of alternating simulations in several points: Since we study connection-safety of assemblies, we do not allow a concrete behaviour to fall silent w.r.t. outputs, when the abstract behaviour showed some output; this condition is inspired by one part of stuck-freedom introduced by Rajamani and Rehof [14] and is represented in clause (5). As we are interested in the communication behaviour of assemblies, we do not abstract from internal actions and use an action τ for invisible behaviour; we require concrete internal actions to exist on the abstract level by clause (3). In contrast, concrete τ actions are optional on the abstract level as long as the iots-simulation relation is taken into account (cf. clause 4). Abstract internal and τ actions are treated by clause (6) and (7) like output actions in clause (5). Finally, and more technically, we remove the requirement of input determinism of interface automata, saying that for each input label there is at most one successor state. Instead, we introduce the weaker condition (8) below, that inputs of the concrete behaviour do not introduce more non-determinism than the corresponding inputs in the abstract behaviour.

Definition 12. *Let $A = ((I_A, O_A, T_A), S_A, s_{0,A}, \Delta_A)$ and $C = ((I_C, O_C, T_C), S_C, s_{0,C}, \Delta_C)$ be iotss such that $I_A \subseteq I_C$, $O_C \subseteq O_A$ and $T_C \subseteq T_A$. A relation $R \subseteq S_A \times S_C$ is an* alternating iots-simulation *for A and C, if for all $(s_A, s_C) \in R$ it holds that*

1. $\forall l \in I_A \,.\, \forall s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta_A \implies (\exists s'_C \in S_C \,.\, (s_C, l, s'_C) \in \Delta_C \wedge (s'_A, s'_C) \in R)$,
2. $\forall l \in O_C \,.\, \forall s'_C \in S_C \,.\, (s_C, l, s'_C) \in \Delta_C \implies (\exists s'_A \in S_A \,.\, (s_A, l, s'_A) \in \hat{\Delta}_A \wedge (s'_A, s'_C) \in R)$,
3. $\forall l \in T_C \,.\, \forall s'_C \in S_C \,.\, (s_C, l, s'_C) \in \Delta_C \implies (\exists s'_A \in S_A \,.\, (s_A, l, s'_A) \in \hat{\Delta}_A \wedge (s'_A, s'_C) \in R)$,
4. $\forall s'_C \in S_C \,.\, (s_C, \tau, s'_C) \in \Delta_C \implies (\exists s'_A \in S_A \,.\, (s_A, \tau, s'_A) \in \hat{\Delta}_A \wedge (s'_A, s'_C) \in R)$,
5. $(\exists l' \in O_A \,.\, \exists s''_A \in S_A \,.\, (s_A, l', s''_A) \in \Delta_A) \implies (\exists l \in O_A \,.\, \exists s'_A \in S_A \,.\, \exists s'_C \in S_C \,.\, (s_A, l, s'_A) \in \Delta_A \wedge (s_C, l, s'_C) \in \Delta_C \wedge (s'_A, s'_C) \in R)$,
6. $(\exists a' \in T_A \,.\, \exists s''_A \in S_A \,.\, (s_A, a', s''_A) \in \Delta_A) \implies (\exists a \in T_A \,.\, \exists s'_A \in S_A \,.\, \exists s'_C \in S_C \,.\, (s_A, a, s'_A) \in \Delta_A \wedge (s_C, a, s'_C) \in \Delta_C \wedge (s'_A, s'_C) \in R)$,
7. $(\exists s''_A \in S_A \,.\, (s_A, \tau, s''_A) \in \Delta_A) \implies (\exists s'_A \in S_A \,.\, \exists s'_C \in S_C \,.\, (s_A, \tau, s'_A) \in \Delta_A \wedge (s_C, \tau, s'_C) \in \Delta_C \wedge (s'_A, s'_C) \in R)$.
8. $\forall l \in I_A \,.\, (\exists s''_A \in S_A \,.\, (s_A, l, s''_A) \in \Delta_A) \implies (\forall s'_C \in S_C \,.\, (s_C, l, s'_C) \in \Delta_C \implies (\exists s'_A \in S_A \,.\, (s_A, l, s'_A) \in \Delta_A \wedge (s'_A, s'_C) \in R))$,

The iots C is a refinement *of the iots A, written $C \sqsubseteq A$, if there exists an alternating iots-simulation R for A and C with $(s_{0,A}, s_{0,C}) \in R$.*

The concept of refinement can be immediately transferred to components by considering their observable behaviours:

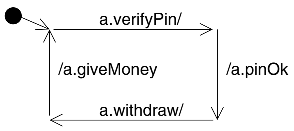


Fig. 10. Refined behaviour of Bank (cf. Fig. 3)

Definition 13. *A component C' is a* subtype *of a component C, written $C \geq C'$, if* $\mathrm{ports}(C) = \mathrm{ports}(C')$ *and $obs(C) \sqsupseteq obs(C')$.*

Example 9 (Refinement). In the context of our Bank–ATM example, the iots in Fig. 10 refines the observable behaviour of the component Bank given in Fig. 3, where the transition of outputting the message that the PIN is not correct has been removed. In particular, the new behaviour is more deterministic than the old one.

6.2 Refinement of Compatible I/O-Transition Systems

Let us first consider refinements of synchronously compatible iotss. In [4], de Alfaro and Henzinger proved that compatibility of input deterministic iots is preserved by their notion of alternating simulations. We extend this result to general iotss and our extended concept of refinement.

Theorem 2. *Let A, B and C be iotss. Let A, B and C, B be composable, and let $A \otimes B$ and $C \otimes B$ be closed. Let A and B be compatible. If $A \sqsupseteq C$, then $A \otimes B \sqsupseteq C \otimes B$ and C and B are compatible.*

Proof. Let $A = ((I_A, O_A, T_A), S_A, s_{0,A}, \Delta_A)$, and similarly for B and C. Let $A \otimes B = ((I_{AB}, O_{AB}, T_{AB}), S_{AB}, s_{0,AB}, \Delta_{AB})$ and $C \otimes B = ((I_{CB}, O_{CB}, T_{CB}), S_{CB}, s_{0,CB}, \Delta_{CB})$. Then $I_{AB} = O_{AB} = I_{CB} = O_{CB} = \emptyset$.

Let R_{AC} be an alternating iots-simulation for A and C with $(s_{0,A}, s_{0,C}) \in R$. 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 have to check clauses (1–8) for alternating iots-simulations for R. Since $I_{AB} = O_{AB} = O_{CB} = \emptyset$, clause (1), clause (2), clause (5) and clause (8) are satisfied vacuously. We only detail clause (3) for $l \in L(A) \bowtie L(B)$; clause (6) for these labels is analogous to (3) and the remaining cases merely transfer the alternating iots-simulation R_{AC} to R.

Let $l \in L(A) \bowtie L(B)$ and $((s_C, s_B), l, (s'_C, s'_B)) \in \Delta_{CB}$. Then $l \in O_A = I_B$ or $l \in I_A = O_B$ and $(s_C, l, s'_C) \in \Delta_C$, $(s_B, l, s'_B) \in \Delta_B$. If $l \in O_A = I_B$, by clause (2) for R_{AC}, there is an $s'_A \in S_A$ with $(s_A, l, s'_A) \in \hat{\Delta}_A$ and thus $((s_A, s_B), l, (s'_A, s'_B)) \in \hat{\Delta}_{AB}$ and also $((s'_A, s'_B), (s'_C, s'_B)) \in R$. If $l \in I_A = O_B$, then, since $(s_A, s_B) \in \mathscr{R}(A \otimes B)$ and A and B are compatible, there is an $s''_A \in S_A$ with $(s_A, l, s''_A) \in \Delta_A$. Thus there is an $s'_A \in S_A$ with $(s_A, l, s'_A) \in \Delta_A$ and $(s'_A, s'_C) \in R_{AC}$ by clause (8). Hence $((s_A, s_B), l, (s'_A, s'_B)) \in \hat{\Delta}_{AB}$ and also $((s'_A, s'_B), (s'_C, s'_B)) \in R$.

In order to show the compatibility of C and B, let $(s_C, s_B) \in \mathscr{R}(C \otimes B)$. Then there is an $(s_A, s_B) \in \mathscr{R}(A \otimes B)$ with $((s_A, s_B), (s_C, s_B)) \in R$, by induction using clause (3) for R. If $(s_C, l, s'_C) \in \Delta_C$ with $l \in O_C \cup T_C$, then there is an $s'_A \in S_A$ with $(s_A, l, s'_A) \in \Delta_A$ by clause (2) for R_{AC} and thus $(s_B, l, s'_B) \in \Delta_B$ for some $s'_B \in S_B$ by the compatibility of A and B. If $(s_B, l, s'_B) \in \Delta_B$ with $l \in O_B = I_A$, then there is an $s'_A \in S_A$ with $(s_A, l, s'_A) \in \Delta_A$ by the compatibility of A and B. Thus there is an $s'_C \in S_C$ with $(s_C, l, s'_C) \in \Delta_C$ by clause (1) for R_{AC}. □

From this theorem we immediately obtain the desired compositionality result for refinement in the case of synchronously compatible iots.

Corollary 1. *Let A, B, C, and D be iotss. Let A, B and C, B and C, D be composable, and let $A \otimes B$, $C \otimes B$, and $C \otimes D$ be closed. Let A and B be compatible. If $A \sqsupseteq C$ and $B \sqsupseteq D$, then $A \otimes B \sqsupseteq C \otimes D$ and C and D are compatible.* □

Example 10 (Refinement and compatibility). The iotss for the behaviours of the Bank–Atm application in Fig. 3 are compatible (modulo port relabelling). The iots in Fig. 10 is a refinement of the original behaviour of the Bank component. By application of Thm. 2 to a synchronous composition of the iotss, we may replace the original Bank behaviour by its refined version and obtain, first, that the composition is a refinement of the original composition and, second, that the refined iots is still (synchronously) compatible with the iots of the Atm behaviour.

For refinements in the context of buffered compatible asynchronous compositions the analogue of Thm. 2 holds. Here, clauses (5), (6) and (7) of Def. 12 for keeping at least one abstract output or internal label in the concrete behaviour are not only conceptually relevant, but also play a major technical role. For technical reasons, we restrict ourselves to a concrete input separated iots and we have to ensure that there are only infinite runs of the composition: An iots $A = (L, S, s_0, \Delta)$ is *deadlock free*, if for all $s \in \mathscr{R}(A)$ there is an $l \in \bigcup L$ and an $s' \in S$ with $(s, l, s') \in \hat{\Delta}$.

Theorem 3. *Let A, B and C be iotss without queue labels. Let A, B and C, B be composable, and let $A \otimes B$ and $C \otimes B$ be closed. Let A and B be buffered compatible, C input separated, and $\Omega(A) \otimes \Omega(B)$ deadlock-free. If $A \sqsupseteq C$, then $\Omega(A) \otimes \Omega(B) \sqsupseteq \Omega(C) \otimes \Omega(B)$ and C and B are buffered compatible.*

Proof. Let $A = (L_A, S_A, s_{0,A}, \Delta_A)$, and similarly for B, C, $\Omega(A)$, $\Omega(B)$, and $\Omega(C)$. Since A, B and C, B are composable, $\Omega(A), \Omega(B)$ and $\Omega(C), \Omega(B)$ are composable by Lem. 3. Let $\Omega(A) \otimes \Omega(B) = ((I_{AB}, O_{AB}, T_{AB}), S_{AB}, q_{0,AB}, \Delta_{AB})$, $\Omega(C) \otimes \Omega(B) = ((I_{CB}, O_{CB}, T_{CB}), S_{CB}, s_{0,CB}, \Delta_{CB})$. By the closedness of $A \otimes B$ and $C \otimes B$, we have $I_A = O_B = I_C$ and $O_A = I_B = O_C$, hence $I_{\Omega(A)} = O_{\Omega(B)} = I_{\Omega(C)}$ and $O_{\Omega(A)} = I_{\Omega(B)} = O_{\Omega(C)}$, and hence $I_{AB} = O_{AB} = I_{CB} = O_{CB} = \emptyset$. Furthermore $T_{AB} = T_{\Omega(A)} \cup L(\Omega(A)) \bowtie L(\Omega(B)) \cup T_{\Omega(B)}$ with $T_A \subseteq T_{\Omega(A)}$.

Let R_{AC} be an alternating simulation for A and C with $(s_{0,A}, s_{0,C}) \in R_{AC}$. Let

$$\begin{aligned} R = \{(((s_A, q), (s_B, q_B)), ((s_C, q), (s_B, q_B))) \mid \\ (s_A, s_C) \in R_{AC} \wedge ((s_A, q), (s_B, q_B)) \in \mathscr{R}(\Omega(A) \otimes \Omega(B))\} . \end{aligned}$$

Then $(((s_{0,A}, \epsilon), (s_{0,B}, \epsilon)), ((s_{0,C}, \epsilon), (s_{0,B}, \epsilon))) \in R$. Let $(((s_A, q), (s_B, q_B)), ((s_C, q), (s_B, q_B))) \in R$. We have to check the clauses (1–8) for alternating iots-simulations for R. Since $I_{AB} = O_{AB} = O_{CB} = \emptyset$, clause (1), clause (2), clause (5) and clause (8) are satisfied vacuously. We only detail clause (3) for $l \in (T_{\Omega(A)} \setminus T_A) \cup (L(\Omega(A)) \bowtie L(\Omega(B)))$; clause (6) for such labels is analogous to (3) and the remaining cases merely transfer the alternating iots-simulation R_{AC} to R.

If $l \in T_{\Omega(A)} \setminus T_A$ and $(((s_C, q), (s_B, q_B)), l, ((s'_C, q'), (s'_B, q'_B))) \in \Delta_{CB}$. Then $l = m?$ for $m \in I_A$, $(s_C, m, s'_C) \in \Delta_C$, $q = m \cdot q'$, and $(s_B, q_B) = (s'_B, q'_B)$. Since $\Omega(A) \otimes \Omega(B)$ is deadlock-free and A and B are buffered compatible, there is an $a \in \bigcup L_A \cup \{\tau\}$ and an $s''_A \in S_A$ with $(s_A, a, s''_A) \in \Delta_A$; but $a \in O_A \cup T_A \cup \{\tau\}$ would contradict the input separation of C at s_C by clauses (5), (6) and (7) for R_{AC}, respectively. Thus $a \in I_A$ and $a = m$, since otherwise A and B would not be buffered compatible. In particular, there is an $s'_A \in S_A$ with $(s_A, m, s'_A) \in \Delta_A$ such that $(s'_A, s'_C) \in R_{AC}$ by clause (1) for R_{AC}. Thus $(((s_A, m \cdot q'), (s_B, q_B)), m?, ((s'_A, q'), (s_B, q_B))) \in \hat{\Delta}_{AB}$ and also $(((s'_A, q'), (s_B, q_B)), ((s'_C, q'), (s_B, q_B))) \in R$.

If $l \in L(\Omega(A)) \bowtie L(\Omega(B))$, then $l = m!$ with either $m \in O_A = I_B$ or $m \in I_A = O_B$. If $m \in O_A = I_B$, then $(s_C, m, s'_C) \in \Delta_C$, $q = q'$, $s_B = s'_B$, $q'_B = q_B \cdot m$. By clause (2) for R_{AC} there is an $s'_A \in S_A$ with $(s_A, m, s'_A) \in \hat{\Delta}_A$ and thus $(((s_A, q), (s_B, q_B)), m!, ((s'_A, q), (s_B, q_B \cdot m))) \in \hat{\Delta}_{AB}$ and also $(((s'_A, q), (s_B, q_B \cdot m)), ((s_C, q), (s_B, q_B \cdot m))) \in R$. If $m \in I_A = O_B$, then $(s_B, m, s'_B) \in \Delta_B$, $q' = q \cdot m$, $s_C = s'_C$, $q'_B = q_B$. Thus $(((s_A, q), (s_B, q_B)), m!, ((s_A, q \cdot m), (s'_B, q_B))) \in \hat{\Delta}_{AB}$ and also $(((s_A, q \cdot m), (s'_B, q_B)), ((s'_C, q \cdot m), (s'_B, q_B))) \in R$.

C and B are also buffered compatible: First, for each infinite run $p_0, l_0, p_1, l_1, \ldots$ of $\Omega(C) \otimes \Omega(B)$ we can inductively construct a simulating run $p'_0, l_0, p'_1, l_1 \ldots$ of $\Omega(A) \otimes \Omega(B)$ such that $(p_k, p'_k) \in R$ for all $k \in \mathbb{N}$: if $(p_k, p'_k) \in R$, then we have $(p_k, l_{k+1}, p_{k+1}) \in \hat{\Delta}_{CB}$ and thus there is a $p'_{k+1} \in S_{AB}$ with $(p'_k, l_{k+1}, p'_{k+1}) \in \hat{\Delta}_{AC}$ and $(p_{k+1}, p'_{k+1}) \in R_{AC}$ by clause (3) for R. Thus, if there would be an infinite weak trace of $\Omega(C) \otimes \Omega(B)$ which is not buffered compatible, there would be an infinite weak trace of $\Omega(A) \otimes \Omega(B)$ which is not buffered compatible, contradicting the buffered compatibility of A and B. $\square$

From Thm. 3 we immediately obtain the desired compositionality result for the refinement of buffered compatible iotss.

Corollary 2. *Let A, B, C, and D be iotss without queue labels. Let A, B and C, B and C, D be composable, and let $A \otimes B$, $C \otimes B$, and $C \otimes D$ be closed. Let C and D be input separated. Let A and B be buffered compatible, and $\Omega(A \otimes B)$ and $\Omega(C \otimes B)$ deadlock-free. If $A \sqsupseteq C$ and $B \sqsupseteq D$, then $\Omega(A) \otimes \Omega(B) \sqsupseteq \Omega(C) \otimes \Omega(D)$ and C and D are buffered compatible.* $\square$

Example 11 (Refinement and buffered compatibility). As an example for the application of Thm. 3 consider the observable behaviours in Fig. 3. Assume again, that the Bank behaviour is refined by an iots as given by Fig. 10. In order to apply the theorem we need to make sure that (1) the iotss in Fig. 3 are buffered compatible, (2) the product of the corresponding buffered iotss is deadlock-free and (3) the refined behaviour is input separated. (1) follows from Ex. 8, (2) is derived from the product in Fig. 4 and (3) is obvious from Fig. 10.

6.3 Substituting Components in Connection-Safe Assemblies

Having shown that refinement preserves (synchronous) compatibility and buffered compatibility, we can finally apply this result to the refinement of components in connection-safe assemblies and show that, under some mild restrictions, connection-safety is preserved when substituting components by refined components. As a technical prerequisite we note that refinement is a pre-congruence w.r.t. relabellings which preserve the kinds of labels:

Lemma 5. *Let A and C be iotss with $L(A) = (I_A, O_A, T_A)$ and $L(C) = (I_C, O_C, T_C)$ and $I_A \subseteq I_C$, $O_C \subseteq O_A$, and $T_A = T_C$. Let $\rho : (I_C, O_A, T_A) \to (I, O, T)$ be a relabelling with $\rho(l_i) \in I$, $\rho(l_o) \in O$, $\rho(l_t) \in T$ for $l_i \in I_C$, $l_o \in O_A$, and $l_t \in T_A$. If $A \sqsupseteq C$, then $A\rho \sqsupseteq C\rho$.* □

Theorem 4. *Let $\langle c : C, d : D; k : K\rangle$ be an assembly with* $\mathrm{ports}(C) = \{p : P\}$ *and* $\mathrm{ports}(D) = \{q : Q\}$ *and $K = (c.p : P, d.q : Q)$. Let C' and D' be components. If k is asynchronous, let $obs(C')$ and $obs(D')$ be input separated and let* $\mathrm{beh}(\langle c : C, d : D; k : K\rangle)$ *and* $\mathrm{beh}(\langle c : C', d : D; k : K\rangle$ *be deadlock-free. If $C \geq C'$, $D \geq D'$ and $\langle c : C, d : D; k : K\rangle$ is connection-safe, then $\langle c : C', d : D; k : K\rangle$ is connection-safe.*

Proof. Let $a = \langle c : C, d : D; k : K\rangle$ and $a' = \langle c : C', d : D'; k : K\rangle$. Then $\mathrm{beh}(a) = obs(c : C)\alpha\sigma \otimes obs(d : D)\alpha\sigma \otimes buf(k : K)$ and $\mathrm{beh}(a') = obs(c : C')\alpha\sigma \otimes obs(d : D')\alpha\sigma \otimes buf(k : K)$. Moreover, $\mathrm{beh}(a)$ and $\mathrm{beh}(a')$ are closed. Let a be connection-safe and $C \geq C'$, $D \geq D'$.

If k is synchronous, then $buf(k : K) = \mathbf{1}$ and α is the identity relabelling. The connection-safety of a thus amounts to the compatibility of $obs(c : C)\sigma$ and $obs(d : D)\sigma$; and a' is connection-safe, if, and only if $obs(c : C')$ and $obs(d : D')$ are compatible. From $C \geq C'$ and $D \geq D'$, we have $obs(C) \sqsupseteq obs(C')$ and $obs(D) \sqsupseteq obs(D')$ and thus $obs(c : C)\sigma \sqsupseteq obs(c : C')\sigma$ and $obs(d : D)\sigma \sqsupseteq obs(d : D')\sigma$ by Lem. 5. From Cor. 1, it follows that $obs(c' : C)$ and $obs(d : D')$ are compatible.

If k is asynchronous, then σ is the identity relabelling and by Prop. 1 we have $\mathrm{beh}(a) = com_{\{c.p:P,d.q:Q\}}(c : C)\kappa \otimes com_{\{c.p:P,d.q:Q\}}(d : D)\kappa$ which is the same as $\Omega_{X_p,Y_p}(obs(c : C))\kappa \otimes \Omega_{X_q,Y_q}(obs(d : D))\kappa$ by Def. 8 with $X_p = \{c.p.m \mid m \in \mathrm{msg}(\mathrm{prv}(P))\}$, and analogously for Y_p, X_q, Y_q. Now $\Omega_{X_p,Y_p}(obs(c : C))\kappa = \Omega(obs(c : C)\kappa)$ and similarly for $d : D$, as C and D have only a single port each and κ is a match relabelling which does not introduce queue labels. Thus the connection-safety of a amounts to the buffered compatibility of $obs(c : C)\kappa)$ and $obs(d : D)\kappa$; and a' is connection-safe, if, and only if $obs(c : C')\kappa$ and $obs(d : D')\kappa)$ are buffered compatible. But $C \geq C'$ and $D \geq D'$ induce $obs(c : C)\kappa \sqsupseteq obs(c : C')\kappa$ and $obs(d : D)\kappa \sqsupseteq obs(d : D')\kappa$ by Lem. 5 and leave $obs(c : C')\kappa$ and $obs(d : D')\kappa$ input separated; hence $obs(c : C')\kappa$ and $obs(d : D')\kappa$ are buffered compatible by Cor. 2. □

Example 12. [Connection-safe assemblies] The behaviours of the Bank–ATM application discussed so far are readily applicable to illustrate Thm. 4. For the implication in Fig. 11 to hold, we need to meet two assumptions in case of an asynchronous connector: first the behaviour of the subtype Bank' must be input separated, i.e. the component

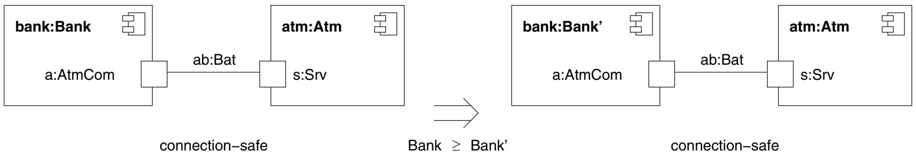


Fig. 11. Subtype substitution preserves connection safety

should behave like a single-threaded system and, second, the original assembly must be deadlock-free. Then connection-safety is preserved when replacing component Bank by its subtype Bank'.

Let Bank and Atm be components with observable behaviours as in Fig. 3. Let Bank' be a component with $\mathrm{ports}(\mathsf{Bank}) = \mathrm{ports}(\mathsf{Bank'})$ and an observable behaviour as in Fig. 10. Then,

- $\langle \mathsf{bank} : \mathsf{Bank}, \mathsf{atm} : \mathsf{Atm}; \mathsf{ab} : \mathsf{Bat}\rangle$ is connection-safe due to Ex. 6 if the connector is synchronous and due to Ex. 8 in the asynchronous case,
- $\mathsf{Bank} \geq \mathsf{Bank'}$ holds due to Ex. 9
- the observable behaviour of Bank' is obviously input separated,
- $\langle \mathsf{bank} : \mathsf{Bank}, \mathsf{atm} : \mathsf{Atm}; \mathsf{ab} : \mathsf{Bat}\rangle$ is deadlock-free, since the product of the corresponding buffered iotss in Fig. 4 is deadlock-free.

Hence $\langle \mathsf{bank} : \mathsf{Bank'}, \mathsf{atm} : \mathsf{Atm}; \mathsf{ab} : \mathsf{Bat}\rangle$ is connection-safe by Thm. 4. $\square$

7 Conclusions

We have presented a component model which supports synchronous and asynchronous communication. For the formal foundation of behaviours we have used I/O-transition systems. The main focus has been on the study of communication behaviours between components in component assemblies. As a crucial desirable property we have required connection-safety of component assemblies which relies on compatibility conditions for iotss with synchronous and asynchronous communication. We have shown that synchronous compatibility is a sufficient criterion for buffered compatibility in asynchronous communications if both communication partners show observable behaviours which are input separated and always eventually inputting. Moreover, we have defined a refinement relation which is compositional w.r.t. synchronous and asynchronous connections of components and which preserves connection-safety.

Our compositionality results are proved for closed systems with only two connected components which already involves a lot of technical efforts due to the formal treatment of asynchronous communication with buffering behaviours. We believe that these results provide a solid basis for an extension of our theorems to closed assemblies with an arbitrary number of components. For the case of open systems further investigation incorporating assumptions on the environment as considered e.g. in [18] is necessary.

References

1. Lau, K.K., Wang, Z.: Software Component Models. IEEE Trans. Softw. Eng. 33(10), 709–724 (2007)
2. Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.): The Common Component Modeling Example. LNCS, vol. 5153. Springer, Heidelberg (2008)
3. Hennicker, R., Janisch, S., Knapp, A.: On the Observable Behaviour of Composite Components. In: Proc. 5th Int. Wsh. Formal Aspects of Component Software, FACS 2008 (2008), `http://www.pst.ifi.lmu.de/veroeffentlichungen/hennicker-et-al:facs:2008.pdf`
4. de Alfaro, L., Henzinger, T.A.: Interface-based Design. In: Broy, M., Grünbauer, J., Harel, D., Hoare, C.A.R. (eds.) Engineering Theories of Software-intensive Systems. NATO Science Series: Mathematics, Physics, and Chemistry, vol. 195, pp. 83–104. Springer, Heidelberg (2005)
5. Bernardo, M., Ciancarini, P., Donatiello, L.: Architecting Families of Software Systems with Process Algebras. ACM Trans. Softw. Eng. Meth. 11(4), 386–426 (2002)
6. Brim, L., Černá, I., Vařeková, P., Zimmerova, B.: Component-Interaction Automata as a Verification-Oriented Component-Based System Specification. SIGSOFT Softw. Eng. Notes 31(2), 1–8 (2006)
7. Brand, D., Zafiropulo, P.: On Communicating Finite-State Machines. J. ACM 30(2), 323–342 (1983)
8. Jéron, T., Jard, C.: Testing for Unboundedness of Fifo Channels. Theo. Comp. Sci. 113, 93–117 (1993)
9. Maréchal, O., Poizat, P., Royer, J.C.: Checking Asynchronously Communicating Components Using Symbolic Transition Systems. In: Meersman, R., Tari, Z. (eds.) OTM 2004. LNCS, vol. 3291, pp. 1502–1519. Springer, Heidelberg (2004)
10. Poizat, P., Royer, J.C.: A Formal Architectural Description Language based on Symbolic Transition Systems and Temporal Logic. J. Univ. Comp. Sci. 12(12), 1741–1782 (2006)
11. Hacklinger, F.: JAVA/A – Taking Components into Java. In: Proc. 13th ISCA Int. Conf. Intelligent and Adaptive Systems and Software Engineering (IASSE 2004), ISCA, pp. 163–169 (2004)
12. Ahumada, S., Apvrille, L., Barros, T., Cansado, A., Madelaine, E., Salageanu, E.: Specifying Fractal and GCM Components with UML. In: 26th Int. Conf. Chilean Computer Science Society (SCCC 2007), pp. 53–62. IEEE, Los Alamitos (2007)
13. Barros, T., Boulifa, R., Madelaine, E.: Parameterized Models for Distributed Java Objects. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 43–60. Springer, Heidelberg (2004)
14. Rajamani, S.K., Rehof, J.: Conformance Checking for Models of Asynchronous Message Passing Software. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 166–179. Springer, Heidelberg (2002)
15. Cécé, G., Finkel, A.: Verification of Programs with Half-Duplex Communication. Inform. Comp. 202(2), 166–190 (2005)
16. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
17. Gouda, M.G., Manning, E.G., Yu, Y.T.: On the Progress of Communications between Two Finite State Machines. Inform. Contr. 63(3), 200–216 (1984)
18. Larsen, K.G., Nyman, U., Wasowski, A.: Interface Input/Output Automata. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 82–97. Springer, Heidelberg (2006)