Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 263 (2010) 47–66
www.elsevier.com/locate/entcs

Behaviour Protocols for Interacting Stateful Components[1]

Sebastian S. Bauer, Rolf Hennicker, Stephan Janisch

Institut für Informatik
Ludwig-Maximilians-Universität München
{bauerse, hennicker, janisch}@pst.ifi.lmu.de

Abstract

We propose a formal foundation for behaviour protocols of interacting, concurrent components with data states. Formally, behaviour protocols are given by labelled transition systems which specify the order of operation invocations as well as the allowed changes of data states of components in terms of pre- and postconditions. We study the compatibility of protocols and we consider their composition which yields a behaviour protocol for a component assembly. Behaviour protocols are equipped with a model-theoretic semantics which describes the class of all correct component or assembly implementations. Implementation models are again formalised in terms of labelled transition systems and the correctness notion is based on an alternating simulation relation between protocol and implementation which takes into account concrete control and data states. As a major result we show that our approach is compositional, i.e. that locally correct implementation models of compatible protocols compose to a globally correct implementation, thus ensuring independent implementability.

Keywords: Behaviour protocol, pre- and postcondition, stateful component, alternating simulation relation, compositionality

1 Introduction

Component-based software development has received much attention not only in practice but also in theory during the last decade. An important role is played by formal specifications of component behaviours which are usually based on a control-flow oriented perspective describing the sequences of actions a component can perform when interacting with its environment; cf., e.g., [16]. Some approaches also consider data that can be transmitted by value passing messages but less attention has been directed towards the integration of *data states* that a component can possess and which are typically specified by invariants and pre- and postconditions. The integrated treatment of control flow and data flow is, however, an important

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

doi:10.1016/j.entcs.2010.05.004

issue in system development, since both aspects appear quite naturally in many business applications. We claim that the combination of the two aspects is far from being well understood. For instance, it is well-known that pre/postcondition style specifications do, in general, not work for systems of concurrent components, but we believe that it is still important to investigate how far one can go by combining both aspects in a concurrent environment. More specifically, this concerns the impact of integrated control flow and data flow on specifications, implementations, formal correctness and compatibility notions, composition, and, last not least, independent implementability.

The working hypothesis of our study is that only on the basis of a precise formal semantics we can capture the subtilities which arise when considering concurrently running components whose interactions have an effect on their data states. Since specifications are inherently loose, leaving freedom to design decisions in implementations, we will follow here the loose semantics approach which, in the spirit of Hoare [10], considers the semantics of a specification as the class of all its correct implementations, also called models of the specification. In such a framework one gets for free notions like consistency, semantic equivalence of specifications etc. Thus it is the general goal of this work to take up these ideas in the context of behaviour protocols for components and their implementations and to provide a solid semantical basis on top of which correctness notions and compatibility concepts can be defined and evaluated. We propose a strict separation of specification and implementation because specifications are considered as contracts describing the mutual assumptions and guarantees from the implementor's *and* from the user's point of view, while implementations have simply to adhere to the implementation requirements induced by a protocol. In order to study implementation correctness, we propose an alternating simulation relation in the spirit of de Alfaro and Henzinger [6] which relates behaviour protocols and implementation models by taking into account properties of data states as well. Our approach is elaborated in a component-oriented setting with distinguished input and output actions which model messages that are received and sent via the ports of components.

When composing systems out of components it is essential that the single components are behaviourally compatible, i.e. work correctly together. In the context of distinguished input and output actions, an important compatibility requirement concerns the question, whether any output issued by one component meets the partner component in a state, where it expects the corresponding input; cf., e.g., [6]. In the context of data states this should also subsume that a caller must satisfy the precondition of the operation provided by the callee and, conversely, that then the callee guarantees the corresponding postcondition. We have formalised these ideas in terms of a behavioural compatibility relation which extends the concept of strong compatibility in [3], and similar notions in [6,12], to take into account data states. As our main result we show that the proposed concepts for protocol compatibility and implementation correctness work smoothly together. This means, implementation models which are locally correct w.r.t. compatible behaviour protocols compose to a correct implementation model of the composed behaviour protocol of a component

assembly. As a consequence, our framework supports independent implementability of behaviour protocols and substitutability of correct implementations.

The paper is organised as follows. After providing some technical preliminaries, we introduce the main structural elements of our component model in Sect. 2. Then, we define behaviour protocols and consider their compatibility in Sect. 3. In Sect. 4, we provide our formal, model-theoretic semantics for behaviour protocols in terms of component (and assembly) implementations and we present our central compositionality result.

Preliminaries. For the specification and implementation of the *control-flow* aspects of component behaviours we will use labelled transition systems. A *labelled transition system* (LTS) $M = (Q, q_0, L, \Delta)$ consists of a set Q of states, an initial state $q_0 \in Q$, a set L of labels, and a transition relation $\Delta \subseteq Q \times L \times Q$.

To deal with the specification and implementation of the *data* aspects of component behaviours we use observer signatures which describe the externally visible data states of components. An *observer signature* Σ_{Obs} consists of a set of (visible) state variables, also called *observers*. A Σ_{Obs}-*data state* $\sigma : \Sigma_{\mathrm{Obs}} \to \mathcal{V}$ assigns to each state variable in Σ_{Obs} a value in some predefined data universe $\mathcal{V}$. The class of all Σ_{Obs}-data states is denoted by $\mathcal{D}(\Sigma_{\mathrm{Obs}})$. For any observer signature Σ_{Obs} and set X of (logical) variables such that $\Sigma_{\mathrm{Obs}} \cap X = \emptyset$, we assume given a set $\mathcal{S}(\Sigma_{\mathrm{Obs}}, X)$ of *state predicates* φ and a set $\mathcal{T}(\Sigma_{\mathrm{Obs}}, X)$ of *transition predicates* π with associated sets $\mathrm{var}_{\log}(\varphi) \subseteq X$ and $\mathrm{var}_{\log}(\pi) \subseteq X$ of (logical) variables. State predicates refer to single states and transition predicates refer to pre- and poststates. We assume that state predicates $\varphi \in \mathcal{S}(\Sigma_{\mathrm{Obs}}, X)$ are equipped with a *satisfaction relation* $\sigma; \rho \vDash \varphi$ for states $\sigma \in \mathcal{D}(\Sigma_{\mathrm{Obs}})$ and valuations $\rho : \mathrm{var}_{\log}(\varphi) \to \mathcal{V}$. Similarly, for transition predicates $\pi \in \mathcal{T}(\Sigma_{\mathrm{Obs}}, X)$ we assume a satisfaction relation $\sigma, \sigma'; \rho \vDash \pi$, for two states $\sigma, \sigma' \in \mathcal{D}(\Sigma_{\mathrm{Obs}})$ and valuations $\rho : \mathrm{var}_{\log}(\pi) \to \mathcal{V}$. We do not fix a particular syntax for observer signatures, observers, and predicates here, neither a particular definition of $\vDash$. How these notations can be instantiated, e.g., in the context of the Object Constraint Language OCL, is shown in [4].

Example 1.1 In our running example we will model a *Turnstile* component located at the entrance of a subway. We assume that the data state of a turnstile can be observed via two visible state variables: *fare* for the actual costs of a trip, and *passed* for the number of persons who have already passed the turnstile. Thus the set $\{\mathit{fare}, \mathit{passed}\}$ constitutes an observer signature for the turnstile, in the following denoted by $\mathit{obs}(\mathit{Turnstile})$. An $\mathit{obs}(\mathit{Turnstile})$-data state is given by an assignment $\sigma : \mathit{obs}(\mathit{Turnstile}) \to \mathcal{V}$, e.g. $\sigma(\mathit{fare}) = 5$, $\sigma(\mathit{passed}) = 100$. For the notation of state and transition predicates we use usual logical expressions where unprimed observer symbols refer to the prestate and primed observer symbols to the poststate of a transition. For instance, $x \geq \mathit{fare}$ is a state predicate with $\mathrm{var}_{\log}(x \geq \mathit{fare}) = \{x\}$ and $\mathit{passed}' = \mathit{passed} + 1$ is a transition predicate which contains no (logical) variable. The satisfaction relation for predicates is defined as expected. For instance, for σ as above and $\rho(x) = 7$, it holds $\sigma; \rho \vDash x \geq \mathit{fare}$, and for the same σ, for σ' with $\sigma'(\mathit{passed}) = 101$ and for an arbitrary ρ, we have $\sigma, \sigma'; \rho \vDash \mathit{passed}' = \mathit{passed} + 1$.

2 Component Model

In this section we summarize the structural concepts of our component model which extends the one in [9] by introducing observer signatures for ports and components. We do, however, not consider hierarchical components here and we make the simplifying assumption that names of ports and components are globally unique.

An *interface* is a pair $(\Sigma_{\mathrm{Obs}}, Op)$ consisting of an observer signature Σ_{Obs} and a set Op of operations. We assume that an *operation* op is of the form $opname(X_{\mathrm{in}})$ where X_{in} is a (possibly empty) set of input variables. We write $\mathrm{var}_{\mathrm{in}}(op)$ to refer to the input variables of an operation op.

Ports are the access points of components where required operations can be called and invocations of provided operations can be received. A *port signature* $(I_{\mathrm{prov}}, I_{\mathrm{req}})$ consists of a provided interface I_{prov} and a required interface I_{req}. When we talk about a port P, we always assume given a *port declaration* $P : \Sigma$ where Σ is a port signature and P is a globally unique port name. We write $prv(P)$ for the provided interface of Σ, $obs_{prv}(P)$ for the observer signature and $opns_{prv}(P)$ for the operations of $prv(P)$. The operations in $opns_{prv}(P)$ are offered at port P and the observer signature $obs_{prv}(P)$ defines the possible observations that can be made at this port (about the data state of its owning component). Symmetrically, we write $req(P)$ for the required interface of Σ, $obs_{req}(P)$ refers to the observer signature and $opns_{req}(P)$ to the operations of $req(P)$. In this case, the operations in $opns_{req}(P)$ are required from components which are connected to P and the observer signature $obs_{req}(P)$ defines which observations are required about the data states of connected components. Thus the idea of the required observer signature $obs_{req}(P)$ is to express which information of another (connected) component is expected to be visible. Therefore, the state variables occurring in the required interface of a port do *not* belong to the component which owns the port but are expected from the provided interface of (the port of) a component to be connected.

Components encapsulate data states and interact with their environment via ports. The data states of a component can only be observed via observers which are determined by the component's observer signature. The access points of a component are given by ports. Formally, a *component signature* $(\Sigma_{\mathrm{Obs}}, (P : \Sigma_P)_{P \in I})$ consists of an observer signature Σ_{Obs} and a finite family of port declarations $P : \Sigma_P$. In the following when we talk about a component C, we always assume given a *component declaration* $C : \Sigma$ where Σ is a component signature and C is a globally unique component name. We write $obs(C)$ to refer to the observer signature and $ports(C)$ to refer to the ports declared in Σ. We require that for all ports $P \in ports(C)$, $obs_{prv}(P) = obs(C)$, i.e. on each port all observers of its owning component are visible. For a port P, $cmp(P)$ denotes its owning component.

For building assemblies we connect ports of components. If P_1 and P_2 are ports whose interfaces match, i.e. $req(P_1) = prv(P_2)$ and $req(P_2) = prv(P_1)$, then $K : \{P_1, P_2\}$ is a (binary) *connector declaration* where K is a globally unique connector name. In the following when we talk about a connector K, we always assume given a connector declaration of the form $K : \{P_1, P_2\}$ and we write $ports(K)$ to refer to

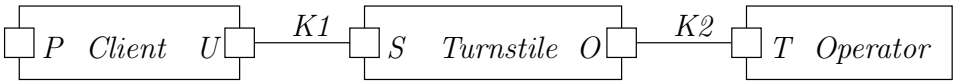

Figure 1. Assembly of the turnstile system.

its set of ports $\{P_1, P_2\}$.

An *assembly* $A = \langle (C : \Sigma_C)_{C\in I}; (K : \{P_1^K, P_2^K\})_{K\in I'}\rangle$ consists of a finite family of component declarations $C : \Sigma_C$ and a finite family of connector declarations $K : \{P_1^K, P_2^K\}$. We write $cmps(A)$ to refer to the components of the assembly, $conns(A)$ to refer to its connectors and we define the set $ports(A) = \bigcup_{C\in cmps(A)} ports(C)$. The open ports of A, i.e., the ports which are not connected, are given by $open(A) = ports(A) \setminus \bigcup_{K\in conns(A)} ports(K)$. For an assembly A, we require that (i) the connectors in $conns(A)$ refer to ports of components inside A, and (ii) there is at most one connector for each port.

Let $\mathcal{A} = (A_\lambda)_{\lambda\in\Lambda}$ be a family of assemblies over a finite set of indices $\Lambda = \{1, \ldots, n\}$. We define $conns(\mathcal{A})$ as the set of all possible connectors between open ports of different assemblies $A_\lambda, A_\kappa \in \mathcal{A}$ with $\lambda \neq \kappa$. A subset $\mathcal{K} \subseteq conns(\mathcal{A})$ is called *valid*, if $\mathcal{K}$ contains at most one connector for each port. We write $ports(\mathcal{K})$ to refer to the set $\bigcup_{K\in\mathcal{K}} ports(K)$. Finally, composition of a family of assemblies $\mathcal{A} = (A_\lambda)_{\lambda\in\Lambda}$ via a valid connector set $\mathcal{K} \subseteq conns(\mathcal{A})$ is denoted by $+_{\mathcal{K}}\mathcal{A}$ and defined by $cmps(\mathcal{A}) = \bigcup_{\lambda\in\Lambda} cmps(A_\lambda)$ and $conns(\mathcal{A}) = \bigcup_{\lambda\in\Lambda} conns(A_\lambda)$. Note that $+_{\mathcal{K}}\mathcal{A}$ satisfies requirements (i) and (ii) again.

Example 2.1 The static structure of a simple simulation system for a turnstile is given by the assembly depicted in Fig. 1. The assembly consists of three components, *Client*, *Turnstile* and *Operator*, and of two connectors *K1:*{*U, S*} and *K2:*{*O, T*}. The observer signature of the *Turnstile* component is given by $obs(Turnstile) = \{fare, passed\}$ as explained in Ex. 1.1. The turnstile has two ports S and O. At port S no observer and no operation is required, i.e. $obs_{req}(S) = \emptyset$ and $opns_{req}(S) = \emptyset$. The port S provides both turnstile observers and two operations: $coin(x)$ for dropping a coin with amount x into the turnstile's slot, and $pass()$ for passing through the turnstile. Formally, the provided interface of port S consists of the observer signature $obs_{prv}(S) = obs(Turnstile)$ and of the set of provided operations $opns_{prv}(S) = \{coin(x), pass()\}$ with $\mathrm{var_{in}}(coin(x)) = \{x\}$ and $\mathrm{var_{in}}(pass()) = \emptyset$.

At port O the turnstile requires no observer but an operation $alarm()$ to inform the operator that a client has tried to pass the turnstile without paying, i.e. formally $obs_{req}(O) = \emptyset$ and $opns_{req}(O) = \{alarm()\}$. The port O provides both turnstile observers and an operation $ready()$ to switch off the alarm mode, i.e. formally $obs_{prv}(O) = obs(Turnstile)$ and $opns_{prv}(O) = \{ready()\}$.

For the other components, *Client* and *Operator*, no data states are considered, i.e. $obs(Client) = obs(Operator) = \emptyset$, and for their ports the following holds: The required (provided) interface of U coincides with the provided (required) interface of S, the required (provided) interface of T coincides with the provided (required) interface of O, and for the open port P we have $obs_{req}(P) = \emptyset$, $opns_{req}(P) = \{ok()\}$, $obs_{prv}(P) = \emptyset$, $opns_{prv}(P) = \emptyset$.

1. Labels $\mathcal{L}^{\mathcal{P}}(P)$ for a port P, $m \in opns_{prv}(P)$, $n \in opns_{req}(P)$:
   - $[\varphi]?P.m[\pi]$ where $\varphi \in \mathcal{S}(obs_{prv}(P), \mathrm{var}_{\mathrm{in}}(m))$, $\pi \in \mathcal{T}(obs_{prv}(P), \mathrm{var}_{\mathrm{in}}(m))$
   - $[\varphi]!P.n[\pi]$ where $\varphi \in \mathcal{S}(obs_{req}(P), \mathrm{var}_{\mathrm{in}}(n))$, $\pi \in \mathcal{T}(obs_{req}(P), \mathrm{var}_{\mathrm{in}}(n))$
2. Labels $\mathcal{L}^{\mathcal{P}}(K)$ for a connector $K : \{P_1, P_2\}$, $m \in opns_{req}(P_i)$, $i, j \in \{1, 2\}$, $i \neq j$:
   - $[\varphi]P_i \triangleright_K P_j.m[\pi]$ where $\varphi \in \mathcal{S}(obs_{req}(P_i), \mathrm{var}_{\mathrm{in}}(m))$, $\pi \in \mathcal{T}(obs_{prv}(P_j), \mathrm{var}_{\mathrm{in}}(m))$

Labels $\mathcal{L}^{\mathcal{P}}(A)$ for an assembly A are given by $\mathcal{L}^{\mathcal{P}}(A) = \bigcup_{P \in open(A)} \mathcal{L}^{\mathcal{P}}(P) \cup \bigcup_{K \in conns(A)} \mathcal{L}^{\mathcal{P}}(K)$ and labels $\mathcal{L}^{\mathcal{P}}(p)$ for a set p of ports are given by $\mathcal{L}^{\mathcal{P}}(p) = \bigcup_{P \in p} \mathcal{L}^{\mathcal{P}}(P)$.

Figure 2. Labels for behaviour protocols.

Restrictions of the component model. In our component model all connectors are binary which would not allow, e.g., broadcasting of messages. In principle, we see no problem to extend our approach to n-ary connectors but, as a matter of fact, the composition of protocols and implementation models discussed later on would then be more involved and the contract principle behind the compatibility notion for protocols must be reworked. For instance, it would be a design decision whether for an n-ary connector with $n > 2$ it would be sufficient to find two partners which can interact according to a protocol or whether full compliance of all n partners would be required.

Concerning data state observers we have required that each port provides all observers of its owning component. Of course, in many examples the observers of a port may only be a subset of the component's observers. The visible state variables (observers) can be considered as simple queries without arguments. In a more general setting one would like to use also n-ary queries as observers. Moreover, we can only connect ports whose interfaces fully agree on required and provided observers and operations. In general, one would wish that the required observers and operations of one port can be strictly included in those of a connected port in the usual contravariant way. These issues concern no serious restrictions and could be resolved at the cost of more involved notations.

A more fundamental restriction is that, unlike [1,9], we do not consider hierarchical components here. The reason is that encapsulating an assembly in a hierarchical component would naturally lead to hidden state variables of components and silent transitions which would need a significant extension of the presented framework.

3 Behaviour Protocols and their Compatibility

Behaviour protocols provide a means to specify the observable behaviour of components and assemblies which integrate the specification of control and data flow behaviour. The transitions in a behaviour protocol are marked with protocol labels which are divided into labels $\mathcal{L}^{\mathcal{P}}(P)$ for ports P and labels $\mathcal{L}^{\mathcal{P}}(K)$ for connectors K; see Fig. 2. Labels for ports model either receiving (?) or sending (!) of a message. Messages which can be received must correspond to operations of the provided interface of a port while messages which can be sent must correspond to operations

of the required interface of a port. Protocol labels are equipped with pre- and postconditions represented by state and transition predicates of the respective observer signatures. More precisely, for a port P, a label $[\varphi]?P.m[\pi]$ models that an operation provided at port P is invoked under the precondition φ and with postcondition π. In this case φ must be a state predicate and π a transition predicate over the observer signature of the *provided* interface of port P, since the operation execution concerns the data state of the owning component of P. Conversely, a label $[\varphi]!P.n[\pi]$ expresses that a message n is sent on port P with precondition φ and postcondition π. Here φ is a state predicate and π a transition predicate over the observer signature of the *required* interface of port P, because the operation execution concerns the data state of a connected component which implements the required operation n. A behaviour protocol for a component whose transitions are labelled with receive labels and/or send labels can be considered as a contract with obligations for both, for the implementor of the component and for a partner component which is going to be connected. The contract principle is detailed in Table 1 where, for a behaviour protocol of a component C, we describe the assumptions that can be made and the guarantees that must be ensured by an implementor and by a communication partner of C. In Table 1 we assume that the component C has a port P with a provided operation m and with a required operation n.

Label	Implementor of C		Partner of C	
	Assume	Guarantee	Assume	Guarantee
$[\varphi_m]?P.m[\pi_m]$	φ_m	π_m	π_m	φ_m
$[\varphi_n]!P.n[\pi_n]$	π_n	φ_n	φ_n	π_n

Table 1
Contract principle of behaviour protocols

Table 1 shows that for an implementor of C a transition with receive label $[\varphi_m]?P.m[\pi_m]$ expresses that the port P must be ready to receive an operation invocation of m under the *assumption* that the precondition φ_m holds, and then the implementor will *guarantee* that after the execution of m the postcondition π_m holds. Conversely, for the correct use of m a partner component must *guarantee* the precondition φ_m if it invokes m, while after the operation execution the partner can *assume* that the postcondition π_m holds.

A send label $[\varphi_n]!P.n[\pi_n]$ imposes the obligation on the implementor of C to *guarantee* that the precondition φ_n is satisfied when the operation n is called while the implementor of C can *assume* that π_n holds when the operation has been executed by the partner. Note, that for the correct use of n in a concrete implementation of C it may be necessary to query first the partner component in order to check whether the precondition φ_n holds in the actual state of the partner; for more details see Sect. 4. Conversely, the partner component (which offers n) can *assume* that upon operation call of n the precondition φ_n is satisfied and the partner must *guarantee* that then, after the execution of n, the postcondition π_n holds.

For a connector K which connects two ports P_i and P_j, a label $[\varphi]P_i \triangleright_K P_j.m[\pi]$ stands for the synchronised sending, reception and execution of an operation m via the connected ports P_i and P_j. In this case the pre- and postconditions are

predicates over the observer signature $obs_{req}(P_i)$, which is the same as $obs_{prv}(P_j)$ since the ports are connected. For an assembly A the protocol labels in $\mathcal{L}^{\mathcal{P}}(A)$ are those labels which correspond to connectors or to open ports of A. Here and in the following all definitions and results are provided for assemblies but they carry over to components since a component C can be identified with a degenerated assembly $\langle\{C\};\emptyset\rangle$ which contains only the component C and no connectors.

Definition 3.1 [Behaviour Protocol]
Let A be an assembly. A *behaviour protocol* for A, also called *A-protocol*, is an LTS $F = (S, s_0, \mathcal{L}^{\mathcal{P}}(A), \Delta)$ where S is a finite set of *protocol states*, $s_0 \in S$ is the *initial protocol state*, $\mathcal{L}^{\mathcal{P}}(A)$ is the set of *protocol labels*, and $\Delta \subseteq S \times \mathcal{L}^{\mathcal{P}}(A) \times S$ is a finite *protocol transition relation*. The class of all A-protocols is denoted by $Prot(A)$.

From the methodological point of view behaviour protocols for a proper assembly A correspond to architecture protocols in [15] while behaviour protocols for components C correspond to frame protocols in [15] and to interface automata in [6] which are supplemented here by pre- and postconditions. In contrast to [6], we do not consider internal transitions of a component here.

Example 3.2 Fig. 3 presents the protocol F_T of the component *Turnstile*. If in the initial state *LOCKED* a coin is deposited whose value is at least the required fare, the turnstile becomes *UNLOCKED*. Then, a person can pass through the turnstile with the effect that the number of passed persons is increased by one and the state *LOCKED* is reached again. If a person tries to pass the turnstile without dropping a coin into its slot an alarm is sent on port O. On the same port the alarm can be shut off by invoking *ready()*. Fig. 4, shows the frame protocol F_C of component *Client* which says that the client may continously call, first *coin(x)* with x being equal to the required *fare* of the turnstile, then *pass()*, and then *ok()* on its open port P. The frame protocol F_O of component *Operator* is shown in Fig. 5. An operator can receive an *alarm()* message after which it can invoke *ready()*.

If a protocol label shows no explicit pre- or postcondition we implicitly assume the trivial predicate *true*. For instance, in Fig. 3 the transition with label $?S.pass()$ between the states *LOCKED* and *ON_ALERT* has the implicit pre- and postcondition *true* while for the same operation called in state *UNLOCKED* the postcondition is $passed' = passed + 1$. Hence, the effect of an invocation of a particular operation may depend on the source control state where the operation is called.

Behaviour protocols can be combined to protocols of assemblies which specify the behaviour of a system of interacting components. For this purpose, we introduce a composition operator $\boxtimes_{\mathcal{K}}$ which composes assembly protocols in accordance with a set $\mathcal{K}$ of valid connectors for a family $\mathcal{A}$ of assemblies. The composition synchronises transitions whose labels match on corresponding inputs and outputs on a connector in $\mathcal{K}$. For instance, if $K : \{P_\lambda, P_\kappa\} \in \mathcal{K}$, a transition with label $[\varphi_\lambda]!P_\lambda.op[\pi_\lambda]$ of one protocol is synchronised with a transition with label $[\varphi_\kappa]?P_\kappa.op[\pi_\kappa]$ of another protocol which results in a transition with label $[\varphi_\lambda \wedge \varphi_\kappa]P_\lambda \triangleright_K P_\kappa.op[\pi_\lambda \wedge \pi_\kappa]$ where the original preconditions (postconditions resp.) are combined by conjunction. Transitions whose labels concern connected ports which cannot be synchronised are

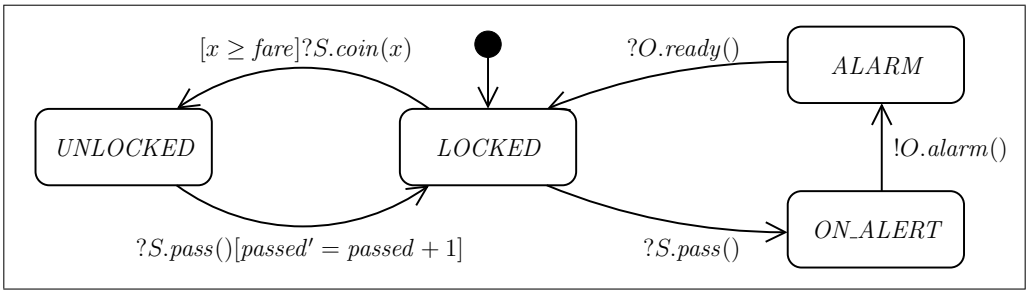

Figure 3. Protocol F_T of component *Turnstile*.

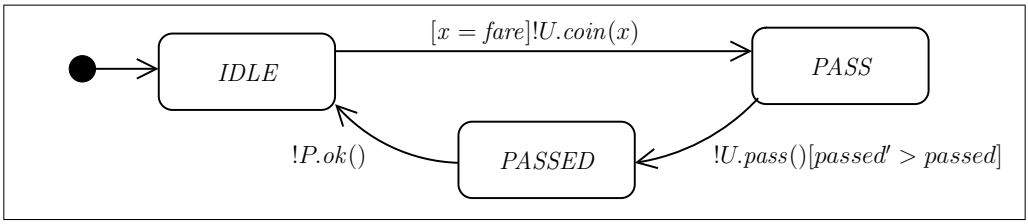

Figure 4. Protocol F_C of component *Client*.

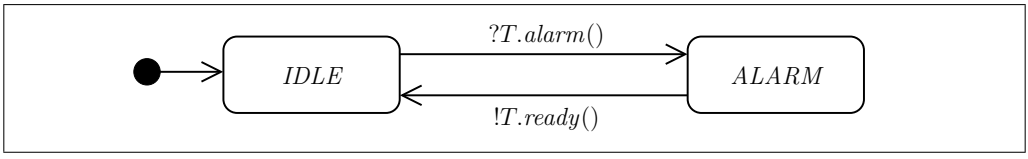

Figure 5. Protocol F_O of component *Operator*.

dropped while transitions whose labels concern unconnected ports are kept in the composition.

Definition 3.3 [Protocol Composition]
Let $\mathcal{A} = (A_\lambda)_{\lambda\in\Lambda}$ be a family of assemblies, let $\mathcal{F} = (F_\lambda)_{\lambda\in\Lambda}$ be a family of protocols $F_\lambda = (S_\lambda, s_{0,\lambda}, \mathcal{L}^{\mathcal{P}}(A_\lambda), \Delta_\lambda) \in \mathit{Prot}(A_\lambda)$, and let $\mathcal{K} \subseteq \mathit{conns}(\mathcal{A})$ be a valid set of connectors. The *protocol composition of* $\mathcal{F}$ *via* $\mathcal{K}$ yields the behaviour protocol $\boxtimes_{\mathcal{K}}\mathcal{F} \in \mathit{Prot}(+_{\mathcal{K}}\mathcal{A})$ defined by

$$\boxtimes_{\mathcal{K}}\mathcal{F} = (S_1 \times \ldots \times S_n, (s_{0,1}, \ldots, s_{0,n}), \mathcal{L}^{\mathcal{P}}(+_{\mathcal{K}}\mathcal{A}), \Delta)$$

where Δ is the least relation satisfying: if $(s_1, \ldots, s_n) \in S_1 \times \ldots \times S_n$, then

(1) for all $K : \{P_\lambda, P_\kappa\} \in \mathcal{K}$, for $i \neq j \in \{\lambda, \kappa\}$,
if $(s_i, [\varphi_i]!P_i.op[\pi_i], s_i') \in \Delta_i$ and $(s_j, [\varphi_j]?P_j.op[\pi_j], s_j') \in \Delta_j$, then
$((s_1, .., s_i, .., s_j, .., s_n), [\varphi_i \wedge \varphi_j]P_i \triangleright_K P_j.op[\pi_i \wedge \pi_j], (s_1, .., s_i', .., s_j', .., s_n)) \in \Delta$;

(2) for all $l_\lambda \in \mathcal{L}^{\mathcal{P}}(A_\lambda) \setminus \mathcal{L}^{\mathcal{P}}(\mathit{ports}(\mathcal{K}))$,
if $(s_\lambda, l_\lambda, s_\lambda') \in \Delta_\lambda$, then $((s_1, .., s_\lambda, .., s_n), l_\lambda, (s_1, .., s_\lambda', .., s_n)) \in \Delta$.

The composition operator is related to the synchronous product of symbolic transition systems defined in [7], where synchronisation vectors are used instead of corresponding input/output labels. While in [7] predicates occurring in labels

always refer to the data states of the owning component, in our case the predicates occurring in send labels refer to the (required) data states of the connected component. Altough this may look as a minor technical discrimination, this reflects a crucial difference of the proposed method. Indeed, referring in required interfaces to the (visible) data states of connected components is a prerequisite for the contract-oriented approach which we follow here.

When composing systems out of components it is important that the single components are behaviourally compatible, i.e. work correctly together. Since we can assume that for components, say C and C', behaviour protocols are given, which both express assumptions and guarantees from their particular viewpoint (cf. Tab. 1), the crucial idea is to ensure that the mutual assumptions and guarantees of the protocols of C and C' fit together. In the context of distinguished input and output actions an important compatibility requirement concerns the question, whether any output issued by one component meets the partner component in a state where it expects the corresponding input; cf., e.g., [6]. In our approach we have to consider this requirement not only from the control flow but also from the data state perspective. In the following, we introduce a compatibility notion for behaviour protocols which extends the concept of strong compatibility in [3] (and similar notions in [6,12]) to take into account data states.

Definition 3.4 [Protocol Compatibility]
Let $\mathcal{A} = (A_\lambda)_{\lambda\in\Lambda}$ be a family of assemblies, let $\mathcal{F} = (F_\lambda)_{\lambda\in\Lambda}$ be a family of protocols $F_\lambda = (S_\lambda, s_{0,\lambda}, \mathcal{L}^{\mathcal{P}}(A_\lambda), \Delta_\lambda) \in \mathit{Prot}(A_\lambda)$, and let $\mathcal{K} \subseteq \mathit{conns}(\mathcal{A})$ be a valid set of connectors. The family $\mathcal{F}$ is $\mathcal{K}$-*compatible* if for all reachable states $(s_1, \ldots, s_n)$ in $\boxtimes_{\mathcal{K}}\mathcal{F}$, for all $K : \{P_\lambda, P_\kappa\} \in \mathcal{K}$, for $i \neq j \in \{\lambda, \kappa\}$ and for all $\sigma \in \mathcal{D}(\mathit{obs}_{req}(P_i))$ and $\rho : \mathrm{var}_{\mathrm{in}}(op) \to \mathcal{V}$ the following holds:
if $(s_i, [\varphi_i]!P_i.op[\pi_i], s_i') \in \Delta_i$ and $\sigma; \rho \vDash \varphi_i$,
then there exists $(s_j, [\varphi_j]?P_j.op[\pi_j], s_j') \in \Delta_j$ such that

(1) $\sigma; \rho \vDash \varphi_j$, and

(2) for all $\sigma' \in \mathcal{D}(\mathit{obs}_{prv}(P_j))$, $(\sigma, \sigma'; \rho \vDash \pi_j$ and $\sigma; \rho \vDash \varphi_i)$ implies $\sigma, \sigma'; \rho \vDash \pi_i$.

Compatibility essentially formalises the requirement that for every (outgoing) operation call specified by a protocol there must exist a corresponding reception, specified by the partner protocol, with compatible preconditions and compatible postconditions. Note that also protocols with a terminal state can be compatible, if all possible outputs specified in one protocol are accepted by the communication partner. More precisely, condition (1) states that whenever for an output label $[\varphi_i]!P_i.op[\pi_i]$ the precondition φ_i is satisfied in some state $\sigma \in \mathcal{D}(\mathit{obs}_{req}(P_i))$ w.r.t. some valuation ρ for the input variables of op, i.e. a caller has fullfilled his *guarantee* w.r.t. σ and ρ, then a callee has a transition with a precondition φ_j which is also satisfied w.r.t. σ and ρ, and hence meets the *assumptions* of the callee. Note that the chosen transition for op on the callee's side may depend on the state σ and on the valuation ρ where the call was issued. Thus condition (1) allows that a precondition of op on the caller's side can be matched by the disjunction of various preconditions which are distributed over different transitions for op on the callee's side. Condition

(2) says that whenever the execution of *op* reaches a state $\sigma' \in \mathcal{D}(obs_{prv}(P_j))$ where the postcondition π_j of *op* (on the chosen transition) is satisfied, i.e. a callee has fulfilled his *guarantee*, then the postcondition π_i of *op*, *assumed* from a caller, is also satisfied in the respective states. Condition (2) also shows, that for the latter implication to hold one can even additionally assume that the precondition φ_i holds w.r.t. σ and ρ, because this has already been guaranteed by the caller before.

For instance, the two protocols shown in Fig. 3 and Fig. 5 are $\{K2\}$-compatible, already without taking into account pre- and postconditions. The protocols in Fig. 3 and Fig. 4 are $\{K1\}$-compatible. In this case, it is essential that the precondition $x = fare$ of *coin(x)* in the *Client* protocol has guaranteed the precondition $x \geq fare$ assumed by the *Turnstile* protocol. Obviously, also the whole family of protocols used in the turnstile example is $\{K1, K2\}$-compatible; note that some states of the turnstile and the operator are not even reachable in the fully composed protocol.

Restrictions of behaviour protocols. First, in a more general setting one would like to specify also data invariants for the states of behaviour protocols which could be integrated in our framework along the lines of [2]. Secondly, in the send labels $[\varphi]!P.op[\pi]$ of a behaviour protocol we have assumed that the precondition φ refers only to the observer signature of the *required* interface of the port P. More generally, it would be possible to admit also observers of the provided interface of P in φ. In this way one could express, e.g., that a message can only be emitted if the values of the state variables of the owning component are in a particular relation to the values of the state variables of the connected component. For notational convenience we have omitted this generalisation here.

Moreover, behaviour protocols do not support silent transitions which would require a major extension, in particular of the notion of compatibility. We believe, however, that this would work on the basis of our weak compatibility notion introduced in [3] without taking into account data states yet.

The transitions in a behaviour protocol model the full execution of an operation and hence may require mutual exclusion in an implementation, thus reducing the degree of possible concurrency. In a more flexible setting one would like to discriminate the events of sending out an operation call, receiving an operation invocation, internal operation execution, sending out the operation result and receiving the result (cf. [15]) and, moreover, to support asynchronous communication and remote resolution of preconditions; cf. [1]. Of course, this flexibility is rather problematic if one wants to trust pre- and postconditions and it is a real challenge to investigate appropriate communication patterns where pre/postcondition reasoning can still be sound in such an environment.

4 Compositional Semantics of Behaviour Protocols

Behaviour protocols are used for the specification of component and assembly behaviours. In order to understand unambiguously the meaning of a specification, it is generally desirable to have a formal semantics of the specification at hand, which can be used as a basis for analysis and further reasoning, concerning, e.g.,

consistency and semantic equivalence of specifications. Since the purpose of any specification formalism is to specify programs and since specifications are inherently loose, leaving freedom to design decisions in implementations, we will follow here the loose semantics approach which, in the spirit of Hoare [10], considers the semantics of a specification as the class of all its correct implementations, also called models of the specification. In order to realise this idea in our setting, we first have to define appropriate mathematical structures that can represent implementations and, secondly, we must define a correctness notion for implementations w.r.t. a given behaviour protocol. Finally, we will prove a compositionality result which relates compositions on the level of protocols (which are of syntactic nature) with compositions on the (semantic) level of implementations. The compositionality result ensures, under the condition of protocol compatibility studied in the last section, the independent implementability of protocols and the substitutability of (correct) implementations.

In order to formalise implementations in our framework, we will use particular labelled transition systems, in the following called implementation models. In contrast to a behaviour protocol, an implementation model describes in a unique way the possible executions of a concurrent program with interleaving semantics. Hence, an implementation model will not be loose like a specification. It cannot be further refined and it is merely abstract in the sense that it provides a compact notation which does not depend on a particular programming language. On the contrary, the implementation models considered here could even be used as a basis to provide a big-step semantics for a subset of concurrent programs written in a concrete programming language, like concurrent Java, similarly as it is done for (a subset) of sequential Java programs in [17]. Though this goes beyond the scope of this paper, it still provides a perspective how to define a formal correctness relation between behaviour protocols and (concurrent) Java programs.

Let us now describe in more detail the construction of implementation models. The states of an implementation model must carry information for both, the control flow and the evolution of data states. To discriminate the two aspects we distinguish between control states and data states. As already explained, a (visible) data state is determined by the values of the observers of a given observer signature. Hence, for a component C, a (visible) data state is an element $\sigma \in \mathcal{D}(obs(C))$. The underlying state space of C in an implementation model is then formed by the cartesian product $Ctrl_C \times \mathcal{D}(obs(C))$ where $Ctrl_C$ is a set of control states. The state space of an assembly is formed by the cartesian product of the state spaces of all its components.

The labels of an implementation model are split into labels $\mathcal{L}(P)$ for ports P and labels $\mathcal{L}(K)$ for connectors K; see Fig. 6. The labels in $\mathcal{L}(P)$ describe incoming and outgoing operation invocations on port P with particular actual parameters. A label of the form $?P.(m, \rho)$ expresses the invocation of a provided operation m on port P with actual parameters determined by a valuation $\rho : \mathrm{var}_{\mathrm{in}}(m) \to \mathcal{V}$. A transition labelled with $?P.(m, \rho)$ connects a state where the operation is called with the state after execution of the operation. Hence the implementation models considered here assume atomic operation executions. A label of the form $(T, !P.(m, \rho))$ expresses

1. Labels $\mathcal{L}(P)$ for a port P, $m \in opns_{prv}(P)$, $n \in opns_{req}(P)$:
   - $?P.(m, \rho)$ where $\rho : \mathrm{var_{in}}(m) \to \mathcal{V}$
   - $(T, !P.(n, \rho))$ where $T \subseteq \mathcal{D}(obs_{req}(P))$, $\rho : \mathrm{var_{in}}(n) \to \mathcal{V}$
2. Labels $\mathcal{L}(K)$ for a connector $K : \{P_1, P_2\}$, $m \in opns_{req}(P_i)$, $i, j \in \{1, 2\}$, $i \neq j$:
   - $P_i \triangleright_K P_j.(m, \rho)$ where $\rho : \mathrm{var_{in}}(m) \to \mathcal{V}$

Labels $\mathcal{L}(A)$ for an assembly A are given by $\mathcal{L}(A) = \bigcup_{P \in open(A)} \mathcal{L}(P) \cup \bigcup_{K \in conns(A)} \mathcal{L}(K)$ and labels $\mathcal{L}(p)$ for a set of ports p are given by $\mathcal{L}(p) = \bigcup_{P \in p} \mathcal{L}(P)$.

Figure 6. Labels for implementation models.

that the implementation sends out an operation call of m with actual parameters determined by ρ provided that a receiver is in some state whose data part matches T. More precisely, T is a set of data states over the required observers which models the fact that the implementation only invokes m if the visible data state of a (potential) receiver component belongs to T. In a concrete program this may require that the sender component performs in an atomic action a test on the data state of a receiver component (by means of the receiver's observers) and, depending on the result, invokes the required operation.

Implementation models for assemblies must also include labels for communication via connectors. For a connector K between ports P_1 and P_2, the set $\mathcal{L}(K)$ consists of labels of the form $P_1 \triangleright_K P_2.(m, \rho)$ which express the synchronised operation invocation (m, ρ) sent on P_1 and received on P_2. The target state of a transition labelled by $P_1 \triangleright_K P_2.(m, \rho)$ is reached when the operation has finished its execution. For an assembly A, the implementation labels in $\mathcal{L}(A)$ are those labels which correspond to connectors or to open ports of A.

Definition 4.1 [Implementation Model]
For an assembly A, an *A-implementation model* (*A-implementation* for short) is an LTS $M = (Q, \boldsymbol{q}_0, \mathcal{L}(A), \Delta)$ with state space

$$Q = \prod_{C \in cmps(A)} Ctrl_C \times \mathcal{D}(obs(C))$$

where for each component $C \in cmps(A)$, $Ctrl_C$ is a set of control states. We require that an implementation model is *input-enabled*, i.e. for each label of the form $?P.(m, \rho) \in \mathcal{L}(A)$ with $P \in open(A)$, and for each state $q \in Q$, there exists a transition $(q, l, q') \in \Delta$. The class of all A-implementations is denoted by $Impl(A)$. For a state $\boldsymbol{q} \in Q$ and a component $C \in cmps(A)$ we write q_C for the projection of $\boldsymbol{q}$ to the state of the component C, which is a pair $q_C = (c, \sigma) \in Ctrl_C \times \mathcal{D}(obs(C))$. We write $\delta(q_C)$ to refer to the data state part σ of q_C.

Actually, we have implemented component assemblies with Java/A [11] making use of the facilities of concurrent Java to realise the dynamic behaviour of implementation models. Outputs are (usually) performed within the thread of a component and inputs are implemented by methods. To ensure the atomicity of the transitions

in implementation models the Java blocking mechanisms must be carefully applied.

Let us now discuss implementation correctness for an implementation model M w.r.t. a given behaviour protocol F. As described in Sect. 3, the protocol F can be considered as a contract between the implementor of a component and its partner. Of course, an implementation model needs only to take into account the assumptions and guarantees as depicted for the implementor of a component in Tab. 1. This means that, for a protocol transition with label $[\varphi_m]?P.m[\pi_m]$, the implementor can assume that the provided operation m is called in accordance with the protocol state *and* with the precondition φ_m and then the implementation must ensure that after the execution of m the postcondition π_m is valid and one can proceed as specified by the protocol. Conversely, for a protocol transition with label $[\varphi_n]!P.n[\pi_n]$, the implementation must ensure that it calls the operation n only in accordance with the protocol state *and* with the precondition φ_n. After the execution of n the implementation can assume the postcondition π_n and it must ensure that one can proceed as specified by the protocol. These considerations can be formalised by adapting the concept of an alternating simulation relation for interface automata in [6] to our needs, where we have to deal with predicates on the specification level and with concrete data states and actual parameters on the implementation level. In this context the alternating simulation relation contains pairs $(s, \boldsymbol{q})$, where s is a protocol state and $\boldsymbol{q}$ is an assembly state; i.e. $\boldsymbol{q}$ determines, for each component C in the assembly, the component's state $q_C = (c, \sigma)$ with control state c and data state σ. The simulation relation is alternating, because reception of messages as specified in the protocol must be simulated in the implementation model, and conversely, every sending of a message in the model must be allowed (i.e. simulated) by the protocol. Thus an implementation may provide more inputs and, conversely, may produce less outputs than the protocol allows.

Definition 4.2 [Alternating Simulation Relation]
Let A be an assembly, $F = (S, s_0, \mathcal{L}^{\mathcal{P}}(A), \Delta_F) \in \mathit{Prot}(A)$ be an A-protocol, and $M = (Q, \boldsymbol{q}_0, \mathcal{L}(A), \Delta_M) \in \mathit{Impl}(A)$. An *alternating simulation relation between F and M* is a relation $R \subseteq S \times Q$ such that for all $(s, \boldsymbol{q}) \in R$,

(1) for all $P \in \mathit{open}(A)$, $C = \mathit{cmp}(P)$,
  (a) for all labels $l = [\varphi]?P.op[\pi] \in \mathcal{L}^{\mathcal{P}}(P)$ and for all $\rho : \mathrm{var}_{\mathrm{in}}(op) \to \mathcal{V}$, if $(s, l, s') \in \Delta_F$ and $\delta(q_C); \rho \vDash \varphi$, then for all transitions $(\boldsymbol{q}, ?P.(op, \rho), \boldsymbol{q}') \in \Delta_M$ it holds $(s', \boldsymbol{q}') \in R$ and $\delta(q_C), \delta(q'_C); \rho \vDash \pi$;
  (b) for all labels $l = (T, !P.(op, \rho)) \in \mathcal{L}(P)$, if $(\boldsymbol{q}, l, \boldsymbol{q}') \in \Delta_M$, then there exists a transition $(s, [\varphi]!P.op[\pi], s') \in \Delta_F$ such that for all $\sigma \in T$ it holds $\sigma; \rho \vDash \varphi$ and $(s', \boldsymbol{q}') \in R$;

(2) for all $K : \{P_1, P_2\} \in \mathit{conns}(A)$, $C_1 = \mathit{cmp}(P_1)$, $C_2 = \mathit{cmp}(P_2)$ it holds: for $i, j \in \{1, 2\}$, $i \neq j$, for all labels $l = P_i \triangleright_K P_j.(op, \rho) \in \mathcal{L}(K)$, if $(\boldsymbol{q}, l, \boldsymbol{q}') \in \Delta_M$, then there exists a transition $(s, [\varphi]P_i \triangleright_K P_j.op[\pi], s') \in \Delta_F$ such that $(s', \boldsymbol{q}') \in R$ and $\delta(q_{C_j}); \rho \vDash \varphi$ and $\delta(q_{C_j}), \delta(q'_{C_j}); \rho \vDash \pi$.

Conditions (1)(a) and (1)(b) formalise the obligations of the implementor of a component according to the contract principle as described above. Condition (2) ex-

presses that communications between components on the implementation level must be allowed, i.e. simulated, by corresponding protocol transitions. Thus, unlike [6], we do not treat communications as invisible (silent) actions, which are abstracted in a refinement relation, because for assembly implementations it is still important that communications conform to the protocol.

Definition 4.3 [Implementation Correctness]
Let A be an assembly, $F = (S, s_0, \mathcal{L}^{\mathcal{P}}(A), \Delta_F) \in \mathit{Prot}(A)$ be an A-protocol, and $M = (Q, \boldsymbol{q}_0, \mathcal{L}(A), \Delta_M) \in \mathit{Impl}(A)$. M is a *correct A-implementation* of F, if there exists an alternating simulation relation R between F and M such that $(s_0, \boldsymbol{q}_0) \in R$. The class of all correct A-implementations of F provides the semantics of F and is denoted by $[\![F]\!]$.

Implementation models can be composed to build larger systems from smaller ones. For this purpose we introduce the operator $\otimes_{\mathcal{K}}$ which composes implementation models in accordance with a set $\mathcal{K}$ of valid connectors for a family $\mathcal{A}$ of assemblies. The composition of implementation models synchronises transitions concerning connected ports if their labels, e.g. $(T, !P_\lambda.(op, \rho))$ and $?P_\kappa.(op, \rho)$ for a connector $K : \{P_\lambda, P_\kappa\} \in \mathcal{K}$, express the same operation invocation (op, ρ) *and* if the caller, say M_λ, meets the callee, say M_κ, in a state whose visible data part lies in the set T which has guarded the send message. Transitions with labels not in $\mathcal{L}(\mathit{ports}(\mathcal{K}))$ concern open ports and are interleaved in the model composition. Note that the model composition expresses the linking of programs in our semantical domain for implementations. If implementation models were not input-enabled one would need, in general, a more involved composition operator and also condition (1a) of the alternating simulation relation must be adjusted.

Definition 4.4 [Model Composition]
Let $\mathcal{A} = (A_\lambda)_{\lambda \in \Lambda}$ be a family of assemblies, let $\mathcal{M} = (M_\lambda)_{\lambda \in \Lambda}$ be a family of implementation models $M_\lambda = (Q_\lambda, \boldsymbol{q}_{0,\lambda}, \mathcal{L}(A_\lambda), \Delta_\lambda) \in \mathit{Impl}(A_\lambda)$, and let $\mathcal{K} \subseteq \mathit{conns}(\mathcal{A})$ be a valid set of connectors. The *model composition of $\mathcal{M}$ via $\mathcal{K}$* yields the implementation model $\otimes_{\mathcal{K}}\mathcal{M} \in \mathit{Impl}(+_{\mathcal{K}}\mathcal{A})$ defined by

$$\otimes_{\mathcal{K}}\mathcal{M} = (Q_1 \times \ldots \times Q_n, (\boldsymbol{q}_{0,1}, \ldots, \boldsymbol{q}_{0,n}), \mathcal{L}(+_{\mathcal{K}}\mathcal{A}), \Delta)$$

where Δ is the least relation satisfying: if $(\boldsymbol{q}_1, \ldots, \boldsymbol{q}_1) \in Q_1 \times \ldots \times Q_n$, then

(1) for all $K : \{P_\lambda, P_\kappa\} \in \mathcal{K}$, for $i \neq j \in \{\lambda, \kappa\}$,
if $(\boldsymbol{q}_i, (T, !P_i.(op, \rho)), \boldsymbol{q}_i') \in \Delta_i$ and $(\boldsymbol{q}_j, ?P_j.(op, \rho), \boldsymbol{q}_j') \in \Delta_j$
and $\delta(q_{j,cmp(P_j)}) \in T$, then
$((\boldsymbol{q}_1, .., \boldsymbol{q}_i, .., \boldsymbol{q}_j, .., \boldsymbol{q}_n), P_i \triangleright_K P_j.(op, \rho), (\boldsymbol{q}_1, .., \boldsymbol{q}_i', .., \boldsymbol{q}_j', .., \boldsymbol{q}_n)) \in \Delta$;

(2) for all $l_\lambda \in \mathcal{L}(A_\lambda) \setminus \mathcal{L}(\mathit{ports}(\mathcal{K}))$,
if $(\boldsymbol{q}_\lambda, l_\lambda, \boldsymbol{q}_\lambda') \in \Delta_\lambda$, then $((\boldsymbol{q}_1, .., \boldsymbol{q}_\lambda, .., \boldsymbol{q}_n), l_\lambda, (\boldsymbol{q}_1, .., \boldsymbol{q}_\lambda', .., \boldsymbol{q}_n)) \in \Delta$.

We are now able to prove our central result which shows that locally correct implementations of compatible protocols can be composed to a globally correct implementation of the composed protocol for the component assembly. As an important consequence this result ensures, under the condition of protocol compatibility,

independent implementability of protocols and substitutability of correct implementations.

Theorem 4.5 *Let $\mathcal{A} = (A_\lambda)_{\lambda\in\Lambda}$ be a family of assemblies, $\mathcal{F} = (F_\lambda)_{\lambda\in\Lambda}$ be a family of A_λ-protocols, $\mathcal{M} = (M_\lambda)_{\lambda\in\Lambda}$ be a family of A_λ-implementations, and let $\mathcal{K} \subseteq conns(\mathcal{A})$ be a valid set of connectors. If $M_\lambda \in [\![F_\lambda]\!]$ for all $\lambda \in \Lambda$ and if $\mathcal{F}$ is $\mathcal{K}$-compatible, then $\otimes_{\mathcal{K}}\mathcal{M} \in [\![\boxtimes_{\mathcal{K}}\mathcal{F}]\!]$.*

Proof For all $\lambda \in \Lambda$, let

$$F_\lambda = (S_\lambda, s_{0,\lambda}, \mathcal{L}^{\mathcal{P}}(A_\lambda), \Delta_{F_\lambda}) \text{ and } M_\lambda = (Q_\lambda, \boldsymbol{q}_{0,\lambda}, \mathcal{L}(A_\lambda), \Delta_{M_\lambda}).$$

By assumption there exist alternating simulation relations $R_\lambda \subseteq S_\lambda \times Q_\lambda$ between F_λ and M_λ. For proving the correctness of $\otimes_{\mathcal{K}}\mathcal{M}$ we must find an alternating simulation relation R between $\boxtimes_{\mathcal{K}}\mathcal{F}$ and $\otimes_{\mathcal{K}}\mathcal{M}$ such that $((s_{0,1}, .., s_{0,n}), (\boldsymbol{q}_{0,1}, .., \boldsymbol{q}_{0,n})) \in R$. We define

$$\begin{aligned} R = \{((s_1, .., s_n), (\boldsymbol{q}_1, .., \boldsymbol{q}_n)) \mid \\ (s_\lambda, \boldsymbol{q}_\lambda) \in R_\lambda \text{ for all } \lambda \in \Lambda \text{ and } (s_1, .., s_n) \text{ reachable in } \boxtimes_{\mathcal{K}} \mathcal{F}\}. \end{aligned}$$

Obviously, $((s_{0,1}, .., s_{0,n}), (\boldsymbol{q}_{0,1}, .., \boldsymbol{q}_{0,n})) \in R$. We prove that R is indeed an alternating simulation relation. Assume $((s_1, .., s_n), (\boldsymbol{q}_1, .., \boldsymbol{q}_n)) \in R$. It is straightforward to see that condition (1) of Def. 4.2 is satisfied for R, since $open(+_{\mathcal{K}}\mathcal{A}) \subseteq \bigcup_{\lambda\in\Lambda} open(A_\lambda)$. Condition (2) for connectors $K' \notin \mathcal{K}$ is also satisfied by assumption. Hence, we only need to consider condition (2) in Def. 4.2 for connectors $K : \{P_\lambda, P_\kappa\} \in \mathcal{K}$: Let $C_\lambda = cmp(P_\lambda), C_\kappa = cmp(P_\kappa)$ and assume, w.l.o.g., a transition

$$((\boldsymbol{q}_1, .., \boldsymbol{q}_\lambda, .., \boldsymbol{q}_\kappa, .., \boldsymbol{q}_n), P_\lambda \triangleright_K P_\kappa.(op, \rho), (\boldsymbol{q}_1, .., \boldsymbol{q}'_\lambda, .., \boldsymbol{q}'_\kappa, .., \boldsymbol{q}_n)) \text{ in } \otimes_{\mathcal{K}}\mathcal{M}.$$

By the rules of model composition, we have

$$(\boldsymbol{q}_\lambda, (T, !P_\lambda.(op, \rho)), \boldsymbol{q}'_\lambda) \in \Delta_{M_\lambda} \text{ and } (\boldsymbol{q}_\kappa, ?P_\kappa.(op, \rho), \boldsymbol{q}'_\kappa) \in \Delta_{M_\kappa}$$

such that the data state $\delta(q_{\kappa,C_\kappa})$ belongs to T. By assumption we know $(s_\lambda, \boldsymbol{q}_\lambda) \in R_\lambda$ and hence, by condition (1b) of Def. 4.2 for R_λ, there exists a transition $(s_\lambda, [\varphi_\lambda]!P_\lambda.op[\pi_\lambda], s'_\lambda) \in \Delta_{F_\lambda}$ such that for all $\sigma \in T$ it holds $\sigma; \rho \vDash \varphi_\lambda$ and $(s'_\lambda, \boldsymbol{q}'_\lambda) \in R_\lambda$; in particular, since $\delta(q_{\kappa,C_\kappa}) \in T$, we have $\delta(q_{\kappa,C_\kappa}); \rho \vDash \varphi_\lambda$. By compatibility (cf. Def. 3.4) there exists a transition $(s_\kappa, [\varphi_\kappa]?P_\kappa.op[\pi_\kappa], s'_\kappa) \in \Delta_{F_\kappa}$ such that $\delta(q_{\kappa,C_\kappa}); \rho \vDash \varphi_\kappa$ and

(*) for all $\sigma' \in \mathcal{D}(obs_{prv}(P_\kappa)), \delta(q_{\kappa,C_\kappa}), \sigma'; \rho \vDash \varphi_\lambda \wedge \pi_\kappa$ implies $\delta(q_{\kappa,C_\kappa}), \sigma'; \rho \vDash \pi_\lambda$.

. By protocol composition it follows that

$$((s_1, .., s_\lambda, .., s_\kappa, .., s_n), [\varphi_\lambda \wedge \varphi_\kappa]P_\lambda \triangleright_K P_\kappa.op[\pi_\lambda \wedge \pi_\kappa], (s_1, .., s'_\lambda, .., s'_\kappa, .., s_n)$$

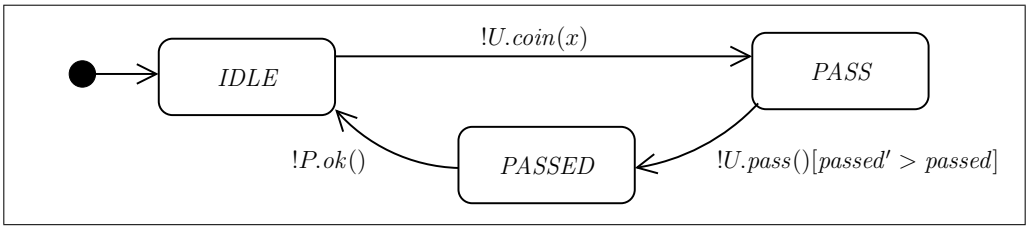

Figure 7. Protocol F'_C of component *Client* which is *not* compatible to F_T in Fig. 3.

is a transition in $\boxtimes_{\mathcal{K}}\mathcal{F}$, and hence $(s_1, .., s'_\lambda, .., s'_\kappa, .., s_n)$ is reachable in $\boxtimes_{\mathcal{K}}\mathcal{F}$. By assumption, we have $(s_\kappa, \boldsymbol{q}_\kappa) \in R_\kappa$, and, moreover, we know $\delta(q_{\kappa,C_\kappa}); \rho \vDash \varphi_\kappa$. Then, it follows from condition (1a) of Def. 4.2 for R_κ that $(s'_\kappa, \boldsymbol{q}'_\kappa) \in R_\kappa$ and $\delta(q_{\kappa,C_\kappa}), \delta(q'_{\kappa,C_\kappa}); \rho \vDash \pi_\kappa$, and by (*) also $\delta(q_{\kappa,C_\kappa}), \delta(q'_{\kappa,C_\kappa}); \rho \vDash \pi_\lambda$ (taking into account $\delta(q_{\kappa,C_\kappa}); \rho \vDash \varphi_\lambda$). Hence, $((s_1, .., s'_\lambda, .., s'_\kappa, .., s_n), (\boldsymbol{q}_1, .., \boldsymbol{q}'_\lambda, .., \boldsymbol{q}'_\kappa, .., \boldsymbol{q}_n)) \in R$ such that $\delta(q_{\kappa,C_\kappa}); \rho \vDash \varphi_\lambda \wedge \varphi_\kappa$ and $\delta(q_{\kappa,C_\kappa}), \delta(q'_{\kappa,C_\kappa}); \rho \vDash \pi_\lambda \wedge \pi_\kappa$. Therefore also condition (2) of Def. 4.2 is satisfied for R. □

The following example illustrates that in the absence of protocol compatibility, Thm. 4.5 would not hold, i.e. independent implementability would not be guaranteed anymore.

Example 4.6 Fig. 7 shows another protocol F'_C for the component *Client*. The difference to the original protocol in Fig. 4 is that the precondition $[x = fare]$ of the call *coin(x)* has been omitted, i.e. changed to *true*. Now the two protocols F'_C and F_T (cf. Fig. 3) are not compatible. Assume correct implementations M_C of F'_C and M_T of F_T such that M_T has an initial data state $\sigma \in \mathcal{D}(obs(Turnstile))$ with $\sigma(fare) = 5$. Moreover, assume that the implementation model M_C first sends out the operation call *coin(3)* which is allowed by F'_C, since there is a corresponding transition in F'_C where the precondition *true* is trivially satisfied. On the other hand, M_T must only follow the protocol transitions in F_T if the precondition of *coin(x)* in F_T is met, otherwise M_T can have an arbitrary behaviour while still remaining a model of F_T. In our example, the expected precondition is not satisfied ($3 \not\geq 5$) when M_C is synchronised with M_T on the operation invocation *coin(3)*. Hence, after the synchronisation M_T could, e.g., issue *alarm()*. This is, however, not an admissible action according to the protocol composition $\boxtimes_{\{KI\}}\{F'_C, F_T\}$. Thus the theorem does not hold without the assumption of compatible protocols.

5 Related Work

Our work is strongly influenced by the semantical considerations in [4,2] and by the concept of alternating simulation relation in [6]. The crucial difference to [6] is that we have integrated (changing) data states and that we have considered different formalisms for specification (finite behaviour protocols) and for implementation (implementation models with possibly infinitely many states). As deviations from [6] we have not considered communications as internal actions, in order to be able to study the conformance of assembly implementations to assembly protocols,

and we have not assumed input-determinism for transition systems but, instead, have used a slightly stronger condition for the preservation of inputs in condition (1a) of Def. 4.2. In contrast to [6] we have only considered protocol compatibility since implementation compatibility is meaningless in the context of input-enabled implementations. In a recent work [14], Mouelhi et al. also consider an extension of the theory of interface automata to the case of data states. Indeed their ideas to deal with protocols are similiar to our treatment but they assume a global set of state variables, they require a stronger condition for compatibility and instead of considering implementation models and implementation correctness, they stay on the level of behaviour protocols and study alternating simulation relations between them. Moreover, their work is not adjusted to a component model with assemblies and connectors yet.

Other related approaches are based on symbolic transition systems (STS) whose labels can be enriched with guards and effect expressions [7,1]. While the approach in [1] is motivated by using STS as finite abstractions of programs for model-checking, the approach of [7] is more directed towards model-checking of symbolic transition systems as specifications and a generative approach to obtain Java code. None of these formalisms focuses on formal correctness notions for implementations and on a contractual interpretation of pre- and postconditions. For instance, there is no compatibility relation between the guards of synchronised transition systems. On the other hand, it would be interesting to see, in what extent the model checking techniques used in [7,1] could be adjusted to check compatibility of components as proposed here.

A methodologically related approach is followed by Černá et al. in [5]. They use component-interaction automata for behavioural modelling of interfaces and define a notion of equivalence on these automata as well as a powerful, parameterised composition operator to prove substitutability and independent implementability properties, similar to our results. The main difference to our work is that they do not consider data states of components. Moreover, refinement and implementation notions in [5] do not follow the idea of alternating simulations where less outputs may be produced by an implementation than allowed by the specification.

Frameworks like CSP-OZ [8] or rCOS [13] deal also with the specification of event-based systems and they take into account data states as well. They do, however, not distinguish (at least semantically) between input and output actions and their expressive power is limited to cases where the effect of an operation on data states is always the same, independent of the control-flow behaviour of a component.

6 Concluding Remarks

We have proposed a formalism for the specification and implementation of component and assembly behaviour in a concurrent environment which integrates the aspects of control flow and evolving data states. The approach is based on a formal semantics for behaviour protocols supporting compositionality and hence independent implementability of components. We clearly separate between specifi-

cations and implementations. The former are given by behaviour protocols which are contract-oriented and express the assumptions and guarantees concerning the implementor and the user of a component. Our approach can be considered as a study of the fundamental concepts that are needed to support the development of concurrent, component-oriented systems when data states of components are taken into account. Hence, on the specification level, our approach is independent from a particular assertion language for pre- and postconditions and, on the implementation level, it is independent from a particular programming language notation. Of course, the instantiation to appropriate subsets of concrete languages, like OCL for assertions, UML for protocols, and concurrent Java for implementations is an interesting objective of further research.

In this paper we have assumed several restrictions concerning the underlying component model and the expressive power of protocols which have been described in the corresponding sections. Some of these restrictions are merely done for notational convenience, others require a significant amount of further research. Under these we consider most important to take into account also silent, internal actions, that may have an effect on the data state of a component, and the investigation of communication patterns which allow more flexibility for concurrent implementations while still ensuring the required assertions for the data states of a component.

Acknowledgement

We would like to thank Moritz Hammer, Alexander Knapp and Michel Bidoit for careful reading of a draft of this paper, Philip Mayer and Andreas Schroeder for their implementations of behaviour protocols with concurrent Java, and the anonymous reviewers of the previous version of this paper for many useful hints and suggestions.

References

[1] Tomás Barros, Rabéa Ameur-Boulifa, Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Behavioural models for distributed fractal components. *Annales des Télécommunications*, 64(1-2):25–43, 2009.

[2] Sebastian S. Bauer and Rolf Hennicker. Views on behaviour protocols and their semantic foundation. In *Proc. CALCO 2009*, volume 5728 of *Lect. Notes Comp. Sci.*, pages 367–382. Springer, 2009.

[3] Sebastian S. Bauer, Philip Mayer, Andreas Schroeder, and Rolf Hennicker. On weak modal compatibility, refinement, and the MIO Workbench. In *Proc. TACAS 2010*, volume 6015 of *Lect. Notes Comp. Sci.*, pages 175–189. Springer, 2010.

[4] Michel Bidoit, Rolf Hennicker, Alexander Knapp, and Hubert Baumeister. Glass-box and black-box views on object-oriented specifications. In *Proc. SEFM'04, Beijing, China*, pages 208–217. IEEE Comp. Society Press, 2004.

[5] Ivana Cerná, Pavlína Vareková, and Barbora Zimmerova. Component substitutability via equivalencies of component-interaction automata. *Electr. Notes Theor. Comput. Sci.*, 182:39–55, 2007.

[6] Luca de Alfaro and Thomas A. Henzinger. Interface-based Design. In Manfred Broy, Johannes Grünbauer, David Harel, and C.A.R. Hoare, editors, *Engineering Theories of Software-intensive Systems*, volume 195 of *NATO Science Series: Mathematics, Physics, and Chemistry*, pages 83–104. Springer, 2005.

[7] Fabrício Fernandes and Jean-Claude Royer. The STSLib project: Towards a formal component model based on STS. *Electr. Notes Theor. Comput. Sci.*, 215:131–149, 2008.

[8] Clemens Fischer. CSP-OZ: a combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Proc. FMOODS*, pages 423–438, Canterbury, UK, 1997. Chapman and Hall, London.

[9] Rolf Hennicker, Stephan Janisch, and Alexander Knapp. On the observable behaviour of composite components. *Electr. Notes Theor. Comput. Sci.*, 260:125–153, 2010.

[10] C.A.R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[11] Alexander Knapp, Stephan Janisch, Rolf Hennicker, Allan Clark, Stephen Gilmore, Florian Hacklinger, Hubert Baumeister, and Martin Wirsing. Modelling the CoCoME with the Java/A component model. In *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lect. Notes Comp. Sci.*, pages 207–237. Springer, 2007.

[12] Kim G. Larsen, Ulrik Nyman, and Andrzej Warsowski. Modal I/O automata for interface and product line theories. In *Proc. ESOP'07*, volume 4421 of *Lect. Notes Comp. Sci.*, pages 64–79. Springer, 2007.

[13] Zhiming Liu, Jifeng He, and Xiaoshan Li. rCOS: Refinement of Component and Object Systems. In *Proc. FMCO 2004*, volume 3657 of *Lect. Notes Comp. Sci.*, pages 183–221. Springer, 2004.

[14] Sebti Mouelhi, Samir Chouali, and Hassan Mountassir. Refinement of interface automata strengthened by action semantics. *Electr. Notes Theor. Comput. Sci.*, 253(1):111–126, 2009.

[15] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Software Eng.*, 28(11):1056–1076, 2002.

[16] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lect. Notes Comp. Sci.* Springer, 2008.

[17] Bernhard Reus, Martin Wirsing, and Rolf Hennicker. A Hoare calculus for verifying Java realizations of OCL-constrained design models. In *Proc. FASE'01*, volume 2029 of *Lect. Notes Comp. Sci.*, pages 300–317. Springer, 2001.