Behaviour Protocols for Interacting Stateful Components

Sebastian S. Bauer, Rolf Hennicker, Stephan Janisch

Abstract

We propose a formal foundation for behaviour protocols of interacting components with (encapsulated) 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 both, control and data states. As a major result we show that our approach is compositional, i.e., that correct implementation models of compatible protocols compose to a correct implementation of the resulting assembly protocol.

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. Thereby 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., [15]. 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. Frameworks like CSP-OZ [8] or rCOS [10] support this facility but they do not distinguish (at least semantically) between input and output actions and their expressive power is limited to cases where the effect of an operation is

---

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

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs
specified independently of the control-flow behaviour of a component. Other approaches are based on symbolic transition systems with a strong focus on finite abstractions for model checking [1,7] but they do not study formal correctness notions to relate specifications and implementations by refinement. The goal of this work is to provide a formal foundation for interacting stateful components with a clear separation between specification and implementation such that the latter can still choose “among the possibilities left by the specification”; see [1]. Of crucial importance is, of course, the study of component composition and preservation of local correctness in global contexts. In this paper we propose an integrated framework which takes into account control flow and the evolution of component data states as well as the discrimination of input and output for distinguishing operation reception and operation call. In particular, we are interested in a contract-based specification style where callee and caller of an operation can formulate its assumptions and guarantees with respect to the data state of the called component. On this basis we study compatibility of specifications, which are called behaviour protocols hereafter.

As a roadmap for this paper we follow the ideas of de Alfaro and Henzinger [5] who have studied interface automata, their refinement, their compatibility and their composition and we aim at an extension of these ideas by taking into account stateful components, where components are equipped with local data states to allow for an evaluation of pre- and postconditions attached to the labels of a behaviour protocol. In the context of data states we cannot use the same formalism for specification and implementation of component behaviour which both have been formalised in terms of interface automata in [5,6]. Though labelled transition systems still provide an appropriate basis, the crucial difference here is that for the purpose of behaviour specification labels must include logical predicates describing pre- and postconditions while for modelling component implementations labels must represent concrete actions like operation invocations which can occur in concrete states of an implementation. As a consequence, we must adjust the alternating simulation relation used by de Alfaro and Henzinger in an appropriate way. Thereby an important role is played by so-called observers which determine the visible states of components.

After providing the necessary technical preliminaries we introduce the main structural elements of our component model in Sect. 2. We study compatibility of behaviour protocols of different components (and assemblies) which are supposed to be composed via connectors between open ports of components (and assemblies) in Sect. 3. Our notion of compatibility extends the corresponding notion in [6] by requiring that if an operation can be called by one component such that the required precondition is satisfied then the connected component must be able to treat the operation invocation in the specified way. We define the (synchronous) composition of behaviour protocols which turns out to have the expected properties (see below) if the protocols are compatible.

In the next step, in Sect. 4, we focus on component (and assembly) implementations which are formalised as labelled transition systems whose states are divided into a control part and a data part and whose transitions carry labels which represent sending or receiving of concrete operation invocations for particular actual parameters. We define a correctness notion for implementations in the spirit of
the alternating simulation relation used for refinements in [6] which requires that any specified input must be accepted by an implementation and that, conversely, any possible output of an implementation must be admitted by the behaviour protocol. In the context of data states the alternating simulation relation between a behaviour protocol and an implementation model is, however, more involved since it has to take into account pre- and postconditions in protocols which must be related to the (visible) data states of component implementations when transitions are performed. Our main theorem shows that the proposed concepts for protocol compatibility and implementation correctness work smoothly together. This means, two implementation models which are locally correct w.r.t. compatible behaviour protocols compose to a correct implementation model of the composed behaviour protocol.

Preliminaries. Our approach is based on labelled transition systems to formalise the control-flow aspects of component behaviours. 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 $\Delta \subseteq Q \times L \times Q$. If $(q, l, q') \in \Delta$, $l$ is enabled at $q$. $M$ is called $L'$-enabled, for a set $L' \subseteq L$ of labels, if for any reachable state $q \in Q$ and label $l \in L'$ there is at least one transition in $\Delta$ such that $l$ is enabled at $q$.

To deal with the specification and implementation of the externally visible data states of components we use observer signatures. An observer signature $\Sigma_{\text{Obs}}$ consists of a set of (visible) state variables, also called observers. A $\Sigma_{\text{Obs}}$-data state $\sigma : \Sigma_{\text{Obs}} \rightarrow V$ assigns to each state variable in $\Sigma_{\text{Obs}}$ a value in some predefined data universe $V$. The class of all $\Sigma_{\text{Obs}}$-data states is denoted by $D(\Sigma_{\text{Obs}})$. For any observer signature $\Sigma_{\text{Obs}}$, we assume given a set $S(\Sigma_{\text{Obs}})$ of state predicates $\varphi$ and a set $T(\Sigma_{\text{Obs}})$ of transition predicates $\pi$ with associated sets $\text{var}(\varphi)$ and $\text{var}(\pi)$ of variables. State predicates refer to single states and transition predicates refer to pre- and poststates. We assume that state predicates $\varphi \in S(\Sigma_{\text{Obs}})$ are equipped with a satisfaction relation $\sigma; \rho \models \varphi$ for states $\sigma \in D(\Sigma_{\text{Obs}})$ and valuations $\rho : \text{var}(\varphi) \rightarrow V$. Similarly, for transition predicates $\pi \in T(\Sigma_{\text{Obs}})$ we assume a satisfaction relation $\sigma, \sigma'; \rho \models \pi$, for two states $\sigma, \sigma' \in D(\Sigma_{\text{Obs}})$ and valuation $\rho : \text{var}(\pi) \rightarrow V$. For $\varphi \in S(\Sigma_{\text{Obs}})$, we write $\models \varphi$ to express that $\varphi$ is universally valid, i.e. $\sigma; \rho \models \varphi$ for all $\sigma \in D(\Sigma_{\text{Obs}})$ and for all valuations $\rho : \text{var}(\varphi) \rightarrow V$. Universal validity of transition predicates is defined in the analogous way. We assume that state predicates (transition predicates resp.) are closed under the usual logical connectives (like $\land$, $\Rightarrow$, etc.) with the usual interpretation. In some occasions we will use state predicates in combination with transition predicates. Then a state predicate $\varphi \in S(\Sigma_{\text{Obs}})$ is considered as a special transition predicate where $\sigma, \sigma'; \rho \models \varphi$ is defined by $\sigma; \rho \models \varphi$. We do not fix a particular syntax for observer signatures, observers, and predicates here; neither a particular definition of $\models$. In the examples unprimed symbols refer to the prestate and primed symbols to the poststate of a transition.

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 sim-
Components interact with each other by using operations which belong to the provided and required interfaces of their ports. An operation op is of the form opname(Xin) where Xin is a (possibly empty) sequence of input variables. We write varin(op) to refer to the input variables of an operation op. An interface is a pair (ΣObs, Op) consisting of an observer signature ΣObs and a set Op of operations. A port signature (Iprov, Ireq) consists of a provided interface Iprov and a required interface Ireq. Throughout this paper when we talk about a port P, we always assume given a port declaration \( P : \Sigma \) where \( \Sigma \) is a port signature and P is a globally unique port name. We write prv(P) for the provided interface of \( \Sigma \), \( obs_{prv}(P) \) 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 \( \Sigma \), \( obs_{req}(P) \) 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 \( obs_{prv}(P) \) defines which observations are required about the data states of connected components.

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 declaration \( C \) is a (possibly empty) sequence of input variables.

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 \( \{P_1, P_2\} \) 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 \( K : \Sigma \) where \( \Sigma \) is a component signature and \( K \) 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 \( \Sigma \). 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.

An assembly \( A = \langle (C : \Sigma_C)_{C \in I}; (K : \{P^K_1, P^K_2\})_{K \in I'} \rangle \) consists of a finite family of component declarations \( C : \Sigma_C \) and a finite family of connector declarations \( K : \{P^K_1, P^K_2\} \). We write \( cmns(A) \) to refer to the components of the assembly, \( conns(A) \) to refer to its connectors and we define \( ports(A) = \bigcup \{ports(C) \mid C \in cmns(A)\} \). The open ports of A, i.e., the ports which are not connected, are given by \( open(A) = ports(A) \setminus \bigcup \{ports(K) \mid K \in conns(A)\} \). For the above definitions

\( \Sigma \)

2 For the sake of simplicity we do not consider output variables here which, however, could be easily integrated in our framework since we follow a synchronous approach with atomic operation executions.

3 In general, the observers of a port could be a subset of the component’s observers. This could be methodologically desirable to emphasize the difference between port and component protocols which, however, goes beyond the scope of this paper. Moreover, we do not consider hidden state variables here, which could be related to the observers of a component via an abstraction function.
to make sense, we require that (i) only ports of components inside \( A \) are connected, i.e., for all \( K \in \text{conns}(A) \) we have that \( \text{ports}(K) \subseteq \text{ports}(A) \); and (ii) there is at most one connector for each port, i.e., if \( P \in \text{ports}(A) \) and \( K, K' \in \text{conns}(A) \) with \( P \in \text{ports}(K) \cap \text{ports}(K') \), then \( K = K' \).

Moreover, we define \( \text{conns}(A_1, A_2) \) as the set of all possible connectors between open ports of assemblies \( A_1 \) and \( A_2 \). A subset \( K \subset \text{conns}(A_1, A_2) \) is called valid if (similar to (ii) above) there is at most one connector for each port. We write \( \text{ports}(K) \) to refer to the set \( \bigcup_{K \in K} \text{ports}(K) \). Finally, composition of two assemblies \( A_1 \) and \( A_2 \) via a valid connector set \( K \subset \text{conns}(A_1, A_2) \) is denoted by \( A = A_1 +_K A_2 \) and defined by \( \text{cmps}(A) = \text{cmps}(A_1) \cup \text{cmps}(A_2) \) and \( \text{conns}(A) = \text{conns}(A_1) \cup \text{conns}(A_2) \cup K \). Note that \( A \) satisfies requirements (i) and (ii) again.

Example 2.1 Our running example models a turnstile located at the entrance of a subway. The static structure of the system is given by the assembly depicted in Fig. 1. It consists of two components, \( \text{Turnstile} \) and \( \text{Operator} \), which are connected via their ports \( O \) and \( T \). The port \( S \) of the turnstile is left open. The observer signatures and the provided and required operations on each port will be explained later when the behaviour protocols of the two components are discussed.

3 Behaviour Protocols and their Compatibility

Behaviour protocols are a means to specify the observable behaviour of components and assemblies.\(^4\) For components they specify the legal sequences of operation invocations on the ports of a component, their invocation conditions and their effect with respect to the (visible) data state of a component. For assemblies behaviour protocols specify, on the one hand, the legal interactions between connected components (taking into account the components’ states) and, on the other hand, the legal sequences of invocations on those ports which are left open in the assembly. We start by introducing the syntax of behaviour protocols which are given by appropriate labelled transition systems.

Protocol labels are divided into labels \( \mathcal{L}^P(P) \) for ports \( P \) and labels \( \mathcal{L}^P(K) \) 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. A label \([\varphi]?P.m[\pi] \) for a port \( P \) expresses that port \( P \) is ready to receive an operation invocation of \( m \) under the assumption that the precondition \( \varphi \) holds, and after the execution of \( m \) the postcondition \( \pi \) is ensured. In this case \( \varphi \) must be a state predicate and \( \pi \) a transition predicate over the observer signature.

\(^4\) We use here the same term as in [14] though our behaviour protocols are based on labelled transition systems.
Labels \( L^P(P) \) for a port, \( m \in \text{ops}_{\text{pre}}(P) \), \( n \in \text{ops}_{\text{req}}(P) \):

- \( [\varphi]|P.m[\pi] \) where \( \varphi \in S(\text{obs}_{\text{pre}}(P)) \), \( \varphi(\varphi) \subseteq \text{var}_{\text{in}}(m) \), \( \pi \in T(\text{obs}_{\text{pre}}(P)) \), \( \varphi(\varphi) \subseteq \text{var}_{\text{in}}(m) \)
- \( [\varphi]|P.n[\pi] \) where \( \varphi \in S(\text{obs}_{\text{req}}(P)) \), \( \varphi(\varphi) \subseteq \text{var}_{\text{in}}(n) \), \( \pi \in T(\text{obs}_{\text{req}}(P)) \), \( \varphi(\varphi) \subseteq \text{var}_{\text{in}}(n) \)

2. Labels \( L^P(K) \) for a connector \( K : \{P_1, P_2\} \), \( m \in \text{ops}_{\text{pre}}(P_j) \), \( i, j \in \{1, 2\}, i \neq j \):

- \( [\varphi]P_i \triangleright_K P_j.m[\pi] \) where \( \varphi \in S(\text{obs}_{\text{pre}}(P_i)) \), \( \varphi(\varphi) \subseteq \text{var}_{\text{in}}(m) \), \( \pi \in T(\text{obs}_{\text{pre}}(P_j)) \), and \( \varphi(\varphi) \subseteq \text{var}_{\text{in}}(m) \)

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

Figure 2: Labels for protocols.

of the provided interface of port \( P \). A label \([\varphi]|P.m[\pi]\) describes the sending of an invocation of \( m \) on port \( P \) with the guarantee of the precondition \( \varphi \) upon operation call and with the expectation that \( \pi \) holds when the operation is finished. Here \( \varphi \) must be a state predicate and \( \pi \) a transition predicate over the observer signature of the required interface of port \( P \). 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 must be predicates over the observer signature \( \text{obs}_{\text{req}}(P_i) \) (which is the same as \( \text{obs}_{\text{pre}}(P_j) \), since required and provided interfaces of connected ports coincide). For an assembly \( A \) the protocol labels in \( L^P(A) \) are those labels which correspond to connectors or to open ports of \( A \).

Definition 3.1 [Behaviour Protocol]

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

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 \( \{\{C\}; \emptyset\} \) which contains only the component \( C \) and no connectors. From the methodological point of view behaviour protocols for a proper assembly \( A \) correspond to architecture protocols in [14] while behaviour protocols for components \( C \) correspond to frame protocols in [14] and to interface automata in [5] which are supplemented here by pre- and postconditions.

Example 3.2 Let us come back to the turnstile example with the assembly shown in Fig. 1. For the observer signature of the Turnstile component we use two visible state variables: fare for the actual costs of a trip, and passed for the number of persons that have already passed the turnstile. The turnstile has two ports \( S \) and \( O \). At port \( S \) no operation is required and two operations are provided: \( \text{coin}(\text{int}: x) \) for dropping a coin with amount \( x \) into the turnstile’s slot, and \( \text{pass()} \) for passing through the turnstile. At port \( O \) the turnstile requires an operation \( \text{alarm}() \) to inform the operator that a client has tried to pass the turnstile without paying and an operation \( \text{ready}() \) is provided to switch off the alarm mode.

Fig. 3 presents protocol \( F_T \) of component Turnstile. If in the initial state
a coin is deposited whose value is at least the required fare the turnstile becomes unlocked. In the state UNLOCKED 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 this causes the turnstile to send out an alarm on its port $O$. On the same port the alarm can be shut off by invoking \texttt{ready}. The frame protocol $F_O$ of component \texttt{Operator} is shown in Fig. 4. It says that an operator can invoke \texttt{ready} whenever an alarm has been received. If a protocol label shows no explicit pre- or postcondition we implicitly assume the trivial predicate $\texttt{true}$. For instance, in the turnstile protocol of Fig. 3 the transition with label $?S$.pass() between the states \texttt{LOCKED} and \texttt{ON_ALERT} has the implicit pre- and postcondition $\texttt{true}$ while for the same operation called in state \texttt{UNLOCKED} the postcondition is $\text{passed}' = \text{passed} + 1$. This shows that the effect of an invocation of a particular operation may indeed depend on the source state where the operation is called which can be conveniently specified by the behaviour protocols introduced here.

![Figure 3. Protocol $F_T$ of component \texttt{Turnstile}.

![Figure 4. Protocol $F_O$ of component \texttt{Operator}.

Two behaviour protocols can be combined to an assembly protocol that describes the behaviour of a system with interacting components. For this purpose we introduce a composition operator $\boxplus_{\mathcal{K}}$ which composes two protocols in accordance with a set of connectors $\mathcal{K}$ between the assemblies underlying the protocols. The composition synchronises transitions whose labels match on corresponding inputs and outputs on a connector in $\mathcal{K}$, given that the pre- and postconditions subsume each other in the (usual) covariant manner where it is allowed to weaken preconditions and to strengthen postconditions; cf., e.g. [11,16]. For instance, if $K : \{P_1, P_2\}$ is a connector in $\mathcal{K}$ such that $P_1$ is a port used in the first protocol and $P_2$ is a port used in the second one, then a transition with label $[\varphi_1]!P_1$.op[$\pi_1$] of the first protocol is synchronised with a transition with label $[\varphi_2]?P_2$.op[$\pi_2$] of the second protocol if it holds that $\varphi_1$ implies $\varphi_2$, and $\varphi_1 \land \pi_2$ implies $\pi_1$. In this case, the synchronisation yields a transition with label $[\varphi_1]P_1 \bowtie_K P_2$.op[$\pi_2$]. The resulting transition expresses a correct communication which can only occur if the pairwise expected and guaranteed pre- and postconditions are satisfied. Thus by protocol composition via $\mathcal{K}$, if $K : \{P_1, P_2\}$ is a connector in $\mathcal{K}$, two transitions with matching labels in $\mathcal{L}^\mathcal{K}(P_1)$
and $\mathcal{L}^P(P_2)$ and semantically matching pre- and postconditions as described above are synchronised to a single transition with label in $\mathcal{L}^P(K)$. Transitions with labels in $\mathcal{L}^P(P_1) \cup \mathcal{L}^P(P_2)$ which can not be matched are disregarded and all other transitions are interleaved in the composition.

**Definition 3.3 [Protocol Composition]**

For $i \in \{1, 2\}$, let $A_i$ be assemblies, $F_i = (S_i, s_{0,i}, \mathcal{L}^P(A_i), \Delta_i) \in \text{Prot}(A_i)$, and let $K \subset \text{conns}(A_1, A_2)$ be a valid set of connectors between $A_1$ and $A_2$. The protocol composition of $F_1$ and $F_2$ via $K$ is defined by

$$F_1 \boxtimes_K F_2 = (S_1 \times S_2, (s_{0,1}, s_{0,2}), \mathcal{L}^P(A_1 +_K A_2), \Delta)$$

where $\Delta$ is the least relation satisfying: if $(s_1, s_2) \in S_1 \times S_2$ then

1. for $i, j \in \{1, 2\}$, $i \neq j$, for all $K : \{P_1, P_2\} \in K$, for all $op \in \text{opns}_{\text{req}}(P_i)$, for all labels $l_i = [\varphi_i] !P_i.\text{op}[\pi_i] \in \mathcal{L}^P(P_i)$ and $l_j = [\varphi_j] ?P_j.\text{op}[\pi_j] \in \mathcal{L}^P(P_j)$, if $(s_1, l_i, s_1') \in \Delta_i$, and $(s_j, l_j, s_j') \in \Delta_j$, and $\models (\varphi_i \Rightarrow \varphi_j)$, and $\models (\varphi_i \land \pi_j \Rightarrow \pi_i)$, then $((s_1, s_2), [\varphi_i]P_i \triangleright_K P_j.\text{op}[\pi_j], (s_1', s_2')) \in \Delta$;
2. for all $l_1 \in \mathcal{L}^P(A_1) \setminus \mathcal{L}^P(\text{ports}(K))$ and $s_2 \in S_2$, if $(s_1, l_1, s_1') \in \Delta_i$ then $((s_1, s_2), l_1, (s_1', s_2)) \in \Delta$;
3. for all $l_2 \in \mathcal{L}^P(A_2) \setminus \mathcal{L}^P(\text{ports}(K))$ and $s_1 \in S_1$, if $(s_2, l_2, s_2') \in \Delta_2$ then $((s_1, s_2), l_2, (s_1, s_2')) \in \Delta$.

The composition operator for protocols is associative (up to equivalence of pre-conditions and of postconditions). It 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 approach we distinguish between send labels, whose predicates refer to the (required) data states of the connected component, and receive labels whose predicates refer to the (provided) data states of the owning component. This is also the difference to [12] where a similar composition operator for protocols has been introduced.

Two protocols are compatible w.r.t. a valid connector set $K$ if, for all connectors $K \in K$, transitions with send labels concerning a port of $K$ can be synchronised with corresponding transitions with a matching input label concerning the other port of $K$. Formally, we define protocol compatibility as follows.

**Definition 3.4 [Protocol Compatibility]**

For $i \in \{1, 2\}$, let $A_i$ be assemblies, $F_i = (S_i, s_{0,i}, \mathcal{L}^P(A_i), \Delta_i) \in \text{Prot}(A_i)$, and let $K \subset \text{conns}(A_1, A_2)$ be a valid set of connectors. $F_1$ and $F_2$ are $K$-compatible if for all reachable states $(s_1, s_2)$ in $F_1 \boxtimes_K F_2$, for $i, j \in \{1, 2\}$, $i \neq j$, for all $K : \{P_1, P_2\} \in K$, for all $l_i = [\varphi_i] !P_i.\text{op}[\pi_i] \in \mathcal{L}^P(P_i)$, if $(s_i, l_i, s_i') \in \Delta_i$ then there exists a transition $(s_j, [\varphi_j] ?P_j.\text{op}[\pi_j], s_j') \in \Delta_j$ such that $\models (\varphi_i \Rightarrow \varphi_j)$ and $\models (\varphi_i \land \pi_j \Rightarrow \pi_i)$.

Compatibility essentially formalises the requirement that for every operation call specified by a protocol there must exist a corresponding reception specified by the other protocol, with the conditions on pre- and postconditions as explained above. Thus it generalises compatibility notions known e.g. from [6,5] by taking into account pre- and postconditions. For instance, the two port protocols shown in
1. Labels $L(P)$ for a port $P$, $m \in \text{opns}_{\text{prv}}(P)$, $n \in \text{opns}_{\text{req}}(P)$:
   - $?P.(m,\rho)$ where $\rho : \text{var}_{\text{in}}(m) \rightarrow V$
   - $(T;!P.(n,\rho))$ where $T \subseteq \mathcal{D}(\text{obs}_{\text{req}}(P))$, $\rho : \text{var}_{\text{in}}(n) \rightarrow V$

2. Labels $L(K)$ for a connector $K : \{P_1, P_2\}$, $m \in \text{opns}_{\text{prv}}(P_j)$, $i, j \in \{1, 2\}$, $i \neq j$:
   - $P_i \bowtie P_j.(m,\rho)$ where $\rho : \text{var}_{\text{in}}(m) \rightarrow V$

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

Figure 5. Labels for implementation models.

Fig. 3 and Fig. 3 are obviously compatible, even without taking into account pre- and postconditions. Another example would be the protocol of a turnstile client starting with a transition with label $[x = \text{fare}]!S.\text{coin}(x)$ followed by a transition with label $!S.\text{pass}()[\text{passed}' > \text{passed}]$. This client protocol would also be compatible with the turnstile protocol in Fig. 3.

4 Protocol Implementation and Compositionality

In this section we define a formal semantics for behaviour protocols in terms of their correct implementations and we study compositionality of implementations and protocols. For the formalisation of implementations we follow a model-theoretic approach where an implementation is represented by an input-enabled labelled transition system, also called implementation model.

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 in the preliminaries 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 $\sigma \in \mathcal{D}(\text{obs}(C))$. The underlying state space of $C$ in an implementation model is then formed by the cartesian product $\text{Ctrl}_C \times Q_C$ of a set of control states $\text{Ctrl}_C$ and a set of data states $Q_C \subseteq \mathcal{D}(\text{obs}(C))$. The state space of an assembly is formed by the cartesian product of the state spaces of all contained components.

We will now define the different labels occurring in implementation models. Implementation labels are divided, like protocol labels, into labels $L(P)$ for ports $P$ and labels $L(K)$ for connectors $K$; see Fig. 5. The labels in $L(P)$ describe the actions of receiving or sending an operation invocation on port $P$. A label of the form $?P.(m,\rho)$ expresses the reception of an invocation of a provided operation $m$ on port $P$ with actual parameters determined by a valuation $\rho : \text{var}_{\text{in}}(m) \rightarrow V$. A transition labelled with $?P.(m,\rho)$ connects the state where the operation is called with the state after execution of the operation. Hence the implementation models considered here assume atomic operation executions. We do not model non terminating programs which could, however, be easily integrated by using partial transitions leading to an undefined state $\perp$. A label of the form $(T;!P.(m,\rho))$ expresses that the implementation sends out an operation call of $m$ with actual parameters $\rho$ provided that the receiver is in some state determined by $T$. More precisely, $T$ is a
set of data states over the observer signature of the connected component which models the fact that the implementation only invokes \( m \) if the visible data state of the receiver component belongs to \( T \). In a concrete implementation this would mean that the sender component performs in an atomic action a test on the data state of the receiver component (by means of the 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 \((m, \rho)\) 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 \( A \). Moreover, the set of input labels \( \mathcal{L}_? (A) \) of \( A \) is given by all labels in \( \mathcal{L}(A) \) of the form \(?P.(m, \rho)\) where \( P \in \text{open}(A) \). Implementation models are required to be \( \mathcal{L}_? (A) \)-enabled, i.e. all provided operations on the open ports of an assembly can be called in each reachable state.

**Definition 4.1 [Implementation Model]**

For an assembly \( A \), an \( A \)-implementation model (\( A \)-implementation for short) is an \( \mathcal{L}_? (A) \)-enabled LTS \( M = (Q, q_0, \mathcal{L}(A), \Delta) \) with state space

\[
Q = \prod_{C \in \text{cmps}(A)} \text{Ctrl}_C \times Q_C
\]

where for each component \( C \in \text{cmps}(A) \), \( \text{Ctrl}_C \) is a set of control states and \( Q_C \subseteq D(\text{obs}(C)) \) is a set of (visible) data states of \( C \). The class of all \( A \)-implementations is denoted by \( \text{Impl}(A) \). For a state \( q \in Q \) and a component \( C \in \text{cmps}(A) \) we write \( q_C \) for the projection of \( q \) to the state of the component \( C \) which is a pair \( q_C = (c, \sigma) \in \text{Ctrl}_C \times Q_C \). We write \( \delta(q_C) \) to refer to the data state part \( \sigma \) of \( q_C \).

We require that implementation models are well-formed: A component’s state may only be changed by a transition whose label involves a port of the component. More specifically, the data state of a component \( C \) can only be changed by a transition whose label has the form \(?P.(m, \rho)\) or \( P_1 \triangleright_K P_2. (m, \rho) \) where \( P \) is a port of \( C \). The formalisation of well-formedness of implementation models is straightforward and omitted here.

Let us now discuss implementation correctness for an implementation model \( M \) w.r.t. a given protocol \( F \). The behaviour protocol \( F \) can be considered as a contract between the implementor and the users of provided and required operations. From the implementor’s point of view this means that it can be assumed that, first, a provided operation is only called where the call is admissible according to the protocol state and, secondly, that the given precondition for the invoked operation is satisfied. Under these assumptions the implementation must ensure that after the execution of the operation the given postcondition is satisfied and that one can proceed as specified by the protocol. If the user does not meet the implementor’s assumptions then the implementation can have an arbitrary behaviour. From the user’s point of view the contract principle imposes the obligation that a required operation is only invoked in accordance with the protocol; in particular the given
(required) precondition must be satisfied. Then the user can assume that the given 
(required) postcondition holds after execution of the operation and that she/he 
can proceed as specified by the protocol. It may still be useful to remark that 
an implementation model plays, in general, the role of an implementor and the 
role of a user. The implementor’s role is shown by transitions with labels of the 
form ?P.(m, ρ) while the user’s role is shown by transitions with labels of the form 
(T, !P.(m, ρ)). The above considerations can be formalised by adapting the concept 
of an alternating simulation relation for interface automata in [5] to our needs where 
we have to deal with predicates on the specification level and with data states on the 
implementation level. In this context the alternating simulation relation contains 
pairs (s, q), where s is a protocol state and q denotes an assembly state; i.e. q 
determines, for each component C in the assembly, the component’s control and 
data state q_C = (c, 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 simulated by 
the protocol. Thus an implementation may provide more inputs (and in fact it does 
so since implementation models are input-enabled) and conversely, implementations 
may produce less outputs than the protocol allows.

Definition 4.2 [Alternating Simulation Relation]
Let A be an assembly, F = (S, s_0, L^P(A), Δ_F) ∈ Prot(A) be an A-protocol, and 
M = (Q, q_0, L(A), Δ_M) ∈ Impl(A). An alternating simulation relation between F 
and M is a relation R ⊆ S × Q such that for all (s, q) ∈ R,

1. for all P ∈ open(A), C = cmp(P),
   a. for all labels l = [ϕ]?P.(op[π]) ∈ L^P(P) and for all ρ : var_{in}(op) → V, 
      if (s, l, s’) ∈ Δ_F and δ(q_C) : ρ ⊨ ϕ then for all transitions (q, ?P.(op, ρ), q’) ∈ 
      Δ_M it holds (s’, q’) ∈ R and δ(q_C’), δ(q’_C) : ρ ⊨ π;
   b. for all labels l = (T, !P.(op, ρ)) ∈ L(P), if (q, l, q’) ∈ Δ_M then there exists a 
      transition (s, [ϕ]P.(op[π]), s’) ∈ Δ_F such that for all σ ∈ T it holds σ; ρ ⊨ ϕ 
      and (s’, q’) ∈ R;

2. for all K : {P_1, P_2} ∈ conns(A), C_1 = cmp(P_1), C_2 = cmp(P_2) it holds:
   for i, j ∈ {1, 2}, i ≠ j, for all labels l = P_i ; v_K P_j.(op, ρ) ∈ L(K), 
   if (q, l, q’) ∈ Δ_M then there exists a transition (s, [ϕ]P_i ; v_K P_j.(op[π]), s’) ∈ Δ_F 
   such that (s’, q’) ∈ R and δ(q_C), δ(q’_C) : ρ ⊨ ϕ and δ(q’_C), δ(q’_C) : ρ ⊨ π.

Conditions (1)(a) and (1)(b) formalise the contract principle behind protocols 
as described above. Condition (2) expresses that communications between components 
on the implementation level must be allowed, i.e. simulated, by corresponding 
protocol transitions. Thus, unlike [6,5] 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. 
At a later stage one can still abstract from communications and build a composite 
component around an assembly which, however, goes beyond the scope of this paper.

We can now define correctness of assembly models w.r.t. assembly protocols in 
terms of alternating simulation relations.

11
Definition 4.3 [Implementation Correctness]
Let $A$ be an assembly, $F = (S, s_0, \mathcal{L}(A), \Delta_F) \in Prot(A)$ be an $A$-protocol, and $M = (Q, q_0, \mathcal{L}(A), \Delta_M) \in Impl(A)$. $M$ is a correct $A$-implementation of $F$, if there exists an alternating simulation relation $R$ between $F$ and $M$ such that $(s_0, q_0) \in R$. The class of all correct $A$-implementations of $F$ provides the semantics of $F$ and is denoted by $\llbracket F \rrbracket$.

Implementation models can be composed to build larger systems from smaller ones. For this purpose we introduce the operator $\otimes$ which composes two implementation models in accordance with a valid connector set $\mathcal{K}$. The composition of two models $M_1$ and $M_2$ synchronises transitions concerning connected ports if their labels, e.g. $(T, !P_1.(op, \rho))$ and $?P_2.(op, \rho)$ for a connector $K : \{P_1, P_2\} \in \mathcal{K}$, express the same operation invocation $(op, \rho)$ and if the caller $M_1$ meets the callee $M_2$ in a state whose visible data part (w.r.t. the connected component) lies in the set $T$ which has guarded the send message. Transitions with labels not in $\mathcal{L}(\text{ports} (\mathcal{K}))$ are interleaved in the model composition.

Definition 4.4 [Model Composition]
For $i \in \{1, 2\}$, let $A_i$ be assemblies, $M_i = (Q_i, q_{0,i}, \mathcal{L}(A_i), \Delta_i) \in Impl(A_i)$, and let $\mathcal{K} \subset \text{conns}(A_1, A_2)$ be a valid set of connectors. The model composition of $M_1$ and $M_2$ via $\mathcal{K}$ is defined by $M_1 \otimes_{\mathcal{K}} M_2 = (Q, q_0, \mathcal{L}(A), \Delta)$ where $A = A_1 +_{\mathcal{K}} A_2$, $Q = Q_1 \times Q_2$, $q_0 = (q_{0,1}, q_{0,2})$, and the transition relation $\Delta$ is the least relation satisfying

1. for all $K : \{P_1, P_2\} \in \mathcal{K}$, for $i, j \in \{1, 2\}$, $i \neq j$, for all $op \in \text{opns}_{\text{req}}(P_i)$, for all $\rho : \text{var}_n(op) \to \mathcal{V}$, if $(q_i, (T, !P_1.(op, \rho)), q_j') \in \Delta_i$ and $(q_j, ?P_2.(op, \rho), q_j') \in \Delta_j$ and $\delta(q_j, \text{comp}(P_j)) \in T$, then $((q_1, q_2), P_1 \triangleright_{\mathcal{K}} P_2.(op, \rho), (q_1', q_2')) \in \Delta$;
2. for all $l_1 \in \mathcal{L}(A_1) \setminus \mathcal{L}(\text{ports} (\mathcal{K}))$ and $q_2 \in Q_2$, if $(q_1, l_1, q_1') \in \Delta_1$ then $((q_1, q_2), l_1, (q_1', q_2)) \in \Delta$;
3. for all $l_2 \in \mathcal{L}(A_2) \setminus \mathcal{L}(\text{ports} (\mathcal{K}))$ and $q_1 \in Q_1$, if $(q_2, l_2, q_2') \in \Delta_2$ then $((q_1, q_2), l_2, (q_1, q_2')) \in \Delta$.

The composition operator $\otimes_{\mathcal{K}}$ is associative and it is straightforward to show that it preserves well-formedness of implementation models.

We are now able to present our centrall compositionality result which says that two correct implementation models with compatible protocols can be composed to a correct implementation model of the composed protocol. Hence the proposed framework supports independent implementability.

Theorem 4.5 For $i \in \{1, 2\}$, let $F_i$ be $A_i$-protocols, $M_i \in \text{Impl}(A_i)$, and let $\mathcal{K} \subset \text{conns}(A_1, A_2)$ be a valid set of connectors. If $M_1 \in \llbracket F_1 \rrbracket$, $M_2 \in \llbracket F_2 \rrbracket$, and $F_1$, $F_2$ are $\mathcal{K}$-compatible, then $M_1 \otimes_{\mathcal{K}} M_2 \in \llbracket F_1 \boxtimes_{\mathcal{K}} F_2 \rrbracket$.

Proof For $i \in \{1, 2\}$, let $F_i = (S_i, s_{0,i}, \mathcal{L}(A_i), \Delta_{F_i})$ be $A_i$-protocols, and $M_i = (Q_i, q_{0,i}, \mathcal{L}(A_i), \Delta_{M_i}) \in \text{Impl}(A_i)$. By assumption there exist alternating simulation relations $R_i \subseteq S_i \times Q_i$ between $F_i$ and $M_i$. For proving the correctness of $M_1 \otimes_{\mathcal{K}} M_2$ we must find an alternating simulation relation $R$ between $F_1 \boxtimes_{\mathcal{K}} F_2$ and $M_1 \otimes_{\mathcal{K}} M_2$.
such that \((s_{0,1}, s_{0,2}), (q_{0,1}, q_{0,2})\) \(\in R\). We define

\[
R = \{((s_1, s_2), (q_1, q_2)) \mid (s_1, q_1) \in R_1 \text{ and } (s_2, q_2) \in R_2 \text{ and } (s_1, s_2) \text{ reachable}\}.
\]

We prove that \(R\) is indeed an alternating simulation relation as required. First, obviously \(((s_{0,1}, s_{0,2}), (q_{0,1}, q_{0,2})) \in R\). Condition (1) of Def. 4.2 is obviously satisfied since open \((A) \subseteq \text{open}(A_1) \cup \text{open}(A_2)\). Condition (2) for connectors \(K' \notin K\) is also satisfied by assumption. Hence, we only need to consider connectors \(K : \{P_1, P_2\} \in K\): Let \(C_1 = \text{cmp}(P_1) \in \text{cmps}(A_1), C_2 = \text{cmp}(P_2) \in \text{cmps}(A_2)\), and assume \(w.l.o.g.\) a transition

\[
((q_1, q_2), P_1 \triangleright_K P_2, (op, \rho), (q_1', q_2')) \text{ in } M_1 \otimes_K M_2.
\]

By the rules of model composition, we have

\[
(q_1, (T, !P_1, (op, \rho)), q_1') \in \Delta_{M_1},
\]

\[
(q_2, (?P_2, (op, \rho)), q_2') \in \Delta_{M_2},
\]

such that \(\delta(q_2, C_2) \in T\). By assumption we know \((s_1, q_1) \in R_1\) and hence by (1) there exists a transition \((s_1, [\varphi_1]!P_1, op[\pi_1], s_1') \in \Delta_F\) such that for all \(\sigma \in T\) it holds \(\sigma; \rho \models \varphi_1\) and \((s_1', q_1') \in R_1\); in particular, \(\delta(q_2, C_2); \rho \models \varphi_1\). By compatibility there exists a transition \((s_2, [\varphi_2]?P_2, op[\pi_2], s_2') \in \Delta_F\) such that \(\models (\varphi_1 \Rightarrow \varphi_2)\) and \(\models (\varphi_1 \land \pi_j \Rightarrow \pi_1)\), hence we get the transition \(([s_1, s_2], [\varphi_1]!P_1 \triangleright_K P_2, op[\pi_2], (s_1', s_2')) \in F_1 \otimes_K F_2\). Since by assumption we have \((s_2, q_2) \in R_2\), it follows from (2) and \(\delta(q_2, C_2); \rho \models \varphi_1\) that \((s_2', q_2') \in R_2\) and \(\delta(q_2, C_2), \delta(q_2', C_2); \rho \models \pi_2\). Hence \(((s_1', s_2'), (q_1', q_2')) \in R\). \(\square\)

The following example illustrates that protocol compatibility is indeed mandatory for the validity of Thm. 4.5.

**Example 4.6** Fig. 6 shows two connected components \(C\) and \(D\) which are specified by the protocols \(F_C\) and \(F_D\) resp. We assume that the state predicate \(\varphi\) is not equivalent to true and does not contain free variables. Then \(F_C\) and \(F_D\) are obviously not compatible because the label \(!P.a\) in \(F_C\) has the (implicit) precondition true. Moreover, extracts of two correct implementations \(M_C \in \llbracket F_C \rrbracket\) and \(M_D \in \llbracket F_D \rrbracket\) are given which show the relevant transitions for this example. In the graphical presentation of \(M_D\) we use the following “relaxed” notations: The indicated initial state denotes a set of reachable states which contains the concrete initial state which we assume does not satisfy \(\varphi\); the transition labelled with \(\{\varphi\}?Q,(a, \rho)\) stands for a set of transitions \(\{ (q, ?Q,(a, \rho), q') \mid \delta(q); \rho \models \varphi \}\) and, similarly, the transition labelled with \(\{\neg\varphi\}?Q,(a, \rho)\) stands for a set of transitions. The implementation model \(M_D\) is correct since for any call of \(a\) on port \(Q\) which meets \(M_D\) in a data state satisfying \(\varphi\) it shows the specified behaviour, otherwise it may show arbitrary behaviour; in our case, \(M_D\) sends out \(b\) on the open port \(R\). In the model composition \(M_C \otimes \{K\} M_D,\) after the synchronisation on \(a\), the message \(b\) will indeed be sent out, because \(M_C\) calls \(a\) whatever the state of \(M_D\) is and the initial state of \(M_D\) does not satisfy \(\varphi\). However, the output of \(b\) is not an admissible action according to the protocol composition \(F_C \otimes \{K\} F_D\) (which anyway is trivial with one state and no transitions). Thus the theorem does not hold without the assumption of compatible protocols.\(\blacksquare\)
5 Related Work

Our work is strongly influenced by the theory of interface automata introduced by de Alfaro and Henzinger in [6,5]. The crucial difference, however, is that we have integrated (changing) data states which led to the consideration of different formalisms for specification (finite behaviour protocols) and for implementation (implementation models with possibly infinitely many states). On this basis we have defined data-state-oriented extensions of the compatibility notion and of the alternating simulation relation of [6] and we have shown that, as desired, the compositionality result of [6] can be lifted to this level. As deviations from [6] we have not considered communications as internal action, in order to be able to study the compliance of assembly implementations to assembly protocols, and we have not assumed input-determinism for transition systems. In contrast to [6] we have only considered protocol compatibility since implementation compatibility is meaningless in the context of input-enabled implementations. As a technical consequence, it was not necessary to restrict the parallel composition operator to compatible states as done in [6]. In a very recent work [12], Mouelhi et al. also study an extension of the theory of interface automata to the case of data states. Indeed their ideas to deal with protocols are quite similar to our treatment but 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 [7,1,2]. While the approach in [1,2] 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, guards for required operations are not used as requirements on the data states of connected components.

A methodologically related approach is followed by Černá et al. in [4]. 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 [4] do not follow the idea of alternating simulations where less outputs may be produced by an implementation than allowed by the specification.

6 Concluding Remarks

We have proposed a formalism for the specification and implementation of component and assembly behaviour which integrates the aspects of control flow and evolving data states. A simpler form of the current approach has been presented in [3] where explicit components and assemblies were not considered and where an implementation can only play one role, either being a user or being the provider of a set of operations. Thus, two different correctness notions were considered in [3], user correctness and implementation correctness, and no alternating simulation was used. Moreover, we have focused in [3] on a sequential approach where the correctness notion for implementations allows to utilise postconditions to infer properties of the connected component’s data states. If, for instance, in two subsequent operation calls the postcondition of the first operation implies the precondition of the second one, the invocation of the latter would be safe in a sequential system even without querying the data state again. Obviously, this would, in general, not work for systems of concurrently running components as considered here, because there might be some interfering operation executions (invoked on different ports) in between possibly changing the component’s data state in an unexpected way. There are essentially three ways out of this problem:

- Blocking of operation execution until the precondition is satisfied (which is suggested in [13] for asynchronous communication but which requires possibly costly deadlock and liveness analysis).
- Querying the data state of the target component to check the precondition before an operation is called; in this case the query and the operation execution must be combined to an atomic action (which is the model supported here at the cost of flexibility and performance).
- Architectural constraints or protocol constraints on access to shared observers (which is still an issue for further investigation).

There are several directions in which the current approach can be further developed. First, it is straightforward to integrate state invariants for components. We hope that we can also incorporate in a straightforward way hierarchically structured components as considered in [9]. A real challenge will be to study to what extent the ideas presented here can be transferred to asynchronously communicating components which use, e.g., FIFO buffers for transmitting messages.

Acknowledgment. We would like to thank Moritz Hammer for careful reading of a draft of this paper and the anonymous reviewers for many valuable hints and suggestions on the previous, submitted version of this paper.
References


