Protocol Specifications and Component Adaptors

DANIEL M. YELLIN and ROBERT E. STROM
IBM T.J. Watson Research Center

In this article we examine the augmentation of application interfaces with enhanced specifications that include sequencing constraints called *protocols*. Protocols make explicit the relationship between messages (methods) supported by the application. These relationships are usually only given implicitly, either in the code or in textual comments. We define notions of interface compatibility based upon protocols and show how compatibility can be checked, discovering a class of errors that cannot be discovered via the type system alone. We then define *software adaptors* that can be used to bridge the difference between applications that have functionally compatible but type- and protocol-incompatible interfaces. We discuss what it means for an adaptor to be *well formed*. Leveraging the information provided by protocols, we show how adaptors can be automatically generated from a high-level description, called an *interface mapping*.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*systems specification methodology*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.4.1 [**Operating Systems**]: Process Management—*concurrency*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Adaptors, interface definition languages, protocol compatibility, protocol conversion, software composition

1. INTRODUCTION

An emerging direction in software is the construction of applications from *components* (often called *parts*). A component defines explicit interfaces through which it can be composed with other components. For instance, a home shopper component, which displays a catalogue and allows customers to place orders, may include a "logging" interface and a "filter" interface. The logging interface can be "connected" to a matching interface in a home financial management component that logs financial records, writes checks, etc. The filter interface can be connected to a component which prunes the catalogue using criteria such as personal preference, supplier location, and price. Composing the home shopper, logging, and filter components by connecting their interfaces would yield a compound component with the collective behavior of its constituent components.

A preliminary version of this article appeared in the ACM OOPSLA 1994 Conference.
Authors' address: P.O. Box 704, Yorktown Heights, NY 10598;
email: {dmy; strom}@ibm.watson.com.

Component-based software is gaining momentum in the commercial world [Udell 1994], as witnessed by vendor implementations of OMG's Corba [Object Management Group 1995] standard (e.g., SOM [IBM 1993]) and other composition technologies, such as OpenDoc [Udell 1994], Ole [Brockschmidt 1994], PARTS [Digitalk 1993], VisualAge [IBM 1994], and Java [Arnold and Gosling 1996].

This article discusses two challenges of component composition: first, how can we specify component interfaces in order to know whether a particular component interface is a valid *mate* of another — that is, that the two components will work properly together if connected? Second, how can we provide *adaptors* to enable component composition when the components are *functionally* compatible, but their visible interfaces are not compatible?

Ideally one needs to look only at the interface specification of a component to understand how to use it. Today's object interfaces, such as those defined by Corba, are based on procedural client-server interfaces. That is, an interface between A and B is defined as a collection of procedures implemented by B and callable by A. We find such interface definitions inadequate because of the following:

—Often there are rules constraining the order in which messages may be sent. For example, an OPEN call may have to precede a READ call. Such rules are not documented in the formal interface specification.

—In many applications, messages might be initiated by either party. In the Corba model, the client always initiates messages, and the server may only send a message to a client in the form of a reply to a call. Of course, a client may send a server a message containing a reference to a callback object, but in that case the two interfaces are defined separately, and it is difficult to formally document semantic constraints relating the two. For instance, there is no way to specify that B can only send a particular message to A after B had received some other message from A. Failure to adhere to the constraint could lead to anomalous behavior such as deadlock.

We therefore adopt augmented interface descriptions containing (1) message (method) signatures both for messages sent and messages received and (2) *protocols* defining the legal sequences of messages that can be exchanged between a component and its mate.

We formulate a notion of *protocol compatibility* such that protocol-compatible objects can be determined to be free of certain important errors that cannot be caught by the type system alone (e.g., deadlock). Our formulation of protocols and protocol compatibility is similar to other recent proposals for adding protocols to interface descriptions (e.g., Nierstrasz [1993] and especially Allen and Garlan [1994]). However, we differ in several important ways. First, unlike some proposals, our protocols are bidirectional, describing both messages that can be sent and messages that can be received by a component. Second, we give a very simple semantics for composability that allows components to be easily checked for protocol compatibility, i.e., it does not require elaborate and computationally expensive machinery such as model-checking technology. Although this has the effect of restricting the number of protocols that are compatible, it makes it much easier to treat protocols as part of the type system and to easily reason about protocols. Third, we do not require a completely new language or environment to implement our scheme.

Instead we provide an implementation strategy enabling one to embed our protocol semantics on top of a variety of languages and environments.

The second question we posed was how to get components to work together when they are functionally compatible but not necessarily type and/or protocol compatible. By functionally compatible we mean that, at a high enough level of abstraction, the service provided by one component is equivalent to the service required by the other; nonetheless, they may realize this functionality via different concrete types and protocols. To get these components to work together we need to produce an adaptor that bridges these incompatibilities.

Because we are willing to tolerate fairly substantial differences between interfaces and because we are looking at generic adaptors that do not have domain-specific knowledge, we cannot produce such adaptors automatically. Instead, we propose a tool that produces an adaptor based upon a declarative description which we call an *interface mapping*. This tool will be able to exploit the protocol as well as the type information in the interfaces. Whereas most other work on software adaptors has stressed techniques for *data type* conversion, our work focuses on the harder problem of *protocol* conversion. In this regard our work is similar to work done on conversion of network protocols, although we have developed a unique formulation and solution to this problem in the more general domain of software adaptors. In this domain different notions, such as parameter relationships, become important issues that have not been adequately addressed by previous work. Adaptors can be viewed as a realization of the notion of *mediators* [Wiederhold 1992].

The main contributions of this article are as follows:

—We provide a simpler semantics of protocol compatibility than traditional techniques. This makes it easier to incorporate protocols into the type system. We also offer implementation techniques to map these semantics to a variety of implementations.

—We differentiate between compatible protocols, which can be bound together using simple composition, and incompatible protocols, where a more sophisticated composition technique (requiring an adaptor) is needed.

—We formally define the notion of a software adaptor that allows the composition of protocol-incompatible components. We show how to check this adaptor for correctness.

—We define a high-level interface mapping language that can be used to automatically synthesize an adaptor for components with incompatible protocols.

In Section 2 we describe our notion of interface specifications, which includes protocols. We show how protocols extend the type system, and we give an algorithm for testing whether two components are protocol compatible. Our work is based upon a synchronous protocol semantics, rather than the usual asynchronous semantics. This allows us to simplify protocol specifications by hiding those messages whose function is to compensate for the asynchronous medium by guaranteeing that both parties to a protocol agree on the order of messages sent and received. Although the synchronous semantics makes it much easier to specify and reason about protocols, it can restrict implementations. For this reason we also describe how to map our semantics onto a variety of implementations.

In Section 3 we focus on the problem of building adaptors to bridge the difference between components with incompatible interfaces. We provide a formal definition of an adaptor between such components and algorithms for deciding whether the adaptor will function correctly as an intermediary between them. Our formulation of adaptors provides additional evidence of the usefulness of adding protocols to interface specifications.

In Section 4 we concentrate on tools to construct adaptors. We introduce the notion of an *interface mapping* between incompatible interface specifications. An interface mapping allows a user to specify the important characteristics required of an adaptor between components containing these interfaces. We describe an algorithm that, given an interface mapping, either produces a well-formed adaptor consistent with the mapping or decides that no such adaptor exists.

In Section 5 we compare our work to related research. In the Appendix we provide those proofs omitted from the body of the article.

2. ENHANCED INTERFACE SPECIFICATIONS

2.1 Protocols

We assume a world of software *components* that interact with other components via typed *interfaces*. Each component exposes one or more interfaces through which messages are sent to and received from a potential *collaboration mate* component. When an interface of component A is *bound* to an interface of component B, they are said to engage in a collaboration: messages sent through A's interface are received at B's interface and vice versa. Each interface has a *type*; this type is associated with a *collaboration specification*, an enhanced interface specification defining the rules governing message exchange.

A component, in general, can expose multiple interfaces, allowing it to simultaneously engage in collaborations with multiple components. Note, however, that any particular collaboration is between exactly two parties. A collaboration between a component C and multiple other components must be modeled by separate interfaces in C, one for each other party it is collaborating with. (See Example 2.2.2.)

Examples of collaborating components (taken from our Global Desktop project [Huynh et al. 1994]) include viewers that can view data on their own but can also collaborate with one another so that corresponding entries are viewed simultaneously, filters that can collaborate with viewers to filter information, and bidders that can collaborate with auctioneers to participate in auctions.

A collaboration specification consists of two parts. The first part, called the *interface signature*, describes the set of messages that can be exchanged between the component and its mate. Besides indicating the type of its parameters, each message in a collaboration specification is labeled as a *send* message or a *receive* message.

The second part of a collaboration specification, called the *protocol*, describes a set of *sequencing constraints*. Sequencing constraints define legal orderings of messages by means of a finite-state grammar. The finite-state grammar is specified by means of a set of named states and a set of transitions, one transition for each message that can be sent or received from a particular state. A transition is of the form

```
<state>: <direction> <message> -> <state>
```

where `<state>` is the symbolic name of a state; `<direction>` is either "+" (receive) or "−" (send); and `<message>` is the name of a message described in the interface signature. Every protocol P has a unique state $init_P$ that is the initial state when the collaboration is established. We do not allow two edges to emanate from a state if those edges have the same label, i.e., if `s1: +M1 -> s2` and `s1: +M2 -> s3` are transitions, then `M1` $\neq$ `M2`. (The same is true if we reverse the + signs to − signs.)

When a collaboration begins, the component's interface will be in the initial state of the protocol. At any point in time, if the interface is in state `s1` then it is permitted to send a message of type `M` only if `s1: -M -> s2` is a transition of the protocol. Alternatively, it may receive a message of type `M` only if `s1:+M -> s2` is a transition. After the message is sent (received), the protocol's state advances to `s2`. Whenever a message is sent or received, the parameters associated with this message type (by the interface signature) are also sent or received. Hence a protocol defines a *finite-state machine*, with edges between states labeled by `+<message>` or `-<message>`. A protocol may have *final states* with no outgoing transitions, or it may be nonterminating. A state is local to an interface; each interface of a component can be in a different state.

A state in which messages can only be sent is called a *send* state. A state in which messages can only be received is called a *receive* state. A state in which the component can either receive or send a message is called a *mixed* state.

We make the following *liveness* assumption: a component will not indefinitely remain in a state that contains a send transition. If it is a send state, or if it is a mixed state but does not receive a message, it will ultimately send a message from that state. Without this assumption, it may be that one party is in a state where it can send message m, and its mate is in a state where it can receive message m, but no progress is made, as the sender never sends the message.

2.2 Examples

In this section we give two examples of collaboration specifications. These examples are implemented in our Global Desktop graphical environment [Huynh et al. 1994]. For presentation purposes we have simplified the specifications to illustrate the main ideas.

Example 2.2.1. The following gives a Filter collaboration specification, describing how a filter component interacts with a data server component. It states that initially the filter is in the state `Stable`. When the filter wants to filter the data (perhaps because the user entered some new filtering criterion), the filter sends the data server a `newFilterRequest` message. After this send, the filter enters the state `Filter`. The mate responds by sending the filter the `itemToBeFiltered` message for each data item it has. Whenever the filter receives such a message, it returns an `ok` message if the item meets the criterion and a `remove` message if it does not. Finally, when the data server has no more data items to check it sends the `noMoreItems` message, causing the filter to return to its `Stable` state.

```
Collaboration Filter {
 Receive Messages {
   itemToBeFiltered(dataItem:ObjectRef);
```

```
    noMoreItems();
  };
  Send Messages {
    newFilterRequest();
    ok();
    remove();
  };
  Protocol {
    States {Stable(init),Filter,Respond};
    Transitions {
     Stable:  -newFilterRequest -> Filter;
     Filter:  +itemToBeFiltered -> Respond;
     Filter:  +noMoreItems      -> Stable;
     Respond  -ok               -> Filter;
     Respond: -remove           -> Filter;
    };
  };
};
```

Example 2.2.2. The following gives an Auctioneer collaboration specification describing how an auctioneer component interacts with a mate component — a bidder — that wants to bid on the auction. We model the auctioneer as having a distinct instance of an interface for each bidder component bound to it. Each such instance has the identical collaboration specification. When the auction begins (or when a bidder attaches to the auction in progress) the auctioneer sends to the bidder a `newItem` message containing information about the current item being auctioned and with an id to use on subsequent bids. The auctioneer then enters state B, representing the fact that this bidder is not the current high bidder for the item. In this state the auctioneer will inform the bidder about new high bids for the item by way of an `update` message, or it will send the bidder an `itemsold` message, indicating that the auction for this item is over. When in state B the auctioneer protocol may also receive a `bid` message from the bidder, in which case it enters state D and evaluates the bid. The auctioneer protocol then responds to the bidder protocol that the bid is either rejected or accepted. In the former case, the auctioneer moves back to state B; in the latter case it moves to state E, representing the fact that this bidder now owns the high bid for the item. From this state the bidder is not expected to bid, but the auctioneer can either inform the bidder of subsequent higher bids for the item received from competing bidders by way of `update` messages, or that the auction is over and that this bidder has bought the item. Each `update` message is expected to be acknowledged by the bidder with an `updateAck` message.

```
Collaboration Auctioneer{
  Receive Messages {
    bid(bidderId:id, itemBiddingOn:string, amount:real);
    updateAck();
  };
  Send Messages {
```

```
      newItem(itemDescr:string,bidderId:id);
      update(highBid:real);
      rejectBid(reason:string);
      acceptBid();
      itemSold();
    };
    Protocol {
     States { A (init), B, C, D, E};
     Transitions {
       A: -newItem        -> B;
       B: -update         -> C;
       B: -itemSold       -> A;
       B: +bid            -> D;
       C: +updateAck      -> B;
       D: -rejectBid      -> B;
       D: -acceptBid      -> E;
       E: -update         -> C;
       E: -itemSold       -> A;
     };
   };
};
```

2.3 Protocol Semantics

When two components collaborate with each other via particular interfaces, each sends and receives messages according to the protocols given by the interface specification. There are two possible semantics one can assign to collaborating components: an *asynchronous* semantics and a *synchronous* semantics. Under the asynchronous semantics, a component may send a message m whenever it is in a state that enables a send m transition, even if its mate is not in a state that enables it to receive that message. Each component in this model has a queue that holds messages that have been sent to it, but which it has not yet received. The asynchronous semantics has been extensively studied [Brand and Zafiropulo 1983; Gouda et al. 1984; 1987]. Although the asynchronous semantics can be easily implemented, it is hard to reason about systems of collaborating components under these semantics; in general, properties of the system such as deadlock are undecidable [Brand and Zafiropulo 1983]. Integrating protocols into the type system of a language under the asynchronous semantics would be difficult.

Under the synchronous semantics, a component C can only send a message m to its mate if C is in a state that enables it to send m and if its mate is in a state that enables it to receive m. The finite-state machines describing the protocols of the two components advance synchronously, so that the sending and receipt of a message are considered an atomic action under this abstraction. No queues are required. Because it is easy to reason about systems of collaborating components under the synchronous semantics, we adopt these semantics. However, we can implement the synchronous semantics without actually requiring the two components to send and receive messages atomically. It turns out that all we really require is that the two components always agree on the *execution trace* — the order of messages

sent and received. This allows us a much greater leeway in how to implement our semantics, while retaining the simplicity in reasoning about protocols. In the rest of this section we make precise the synchronous semantics of collaboration and define what we mean by protocol compatibility. We also give an algorithm that tests for protocol compatibility. In Section 2.5 we elaborate on implementation techniques.

We denote the states of a protocol P by $States(P)$ and its transitions by $Transitions(P)$. For two collaborating protocols P_1 and P_2, we assume that the names of the messages sent from P_1 to P_2 are disjoint from the names of the messages sent from P_2 to P_1. This allows us to write a transition as $s : m \rightarrow s'$, omitting the + and - signs from the message m. The auxiliary function $Polarity(P, m)$ will be positive (+) if m is a receiving message in protocol P and will be negative (-) otherwise.

A *collaboration state* for protocols P_1 and P_2 is a pair $\langle s, t \rangle$, where $s \in States(P_1)$ and $t \in States(P_2)$. A *collaboration history* for protocols P_1 and P_2 is a possibly infinite sequence of the form $\alpha_1 \rightarrow_{m_1} \alpha_2 \rightarrow_{m_2} \cdots$ where

—each α_i is a collaboration state for P_1 and P_2,

—$\alpha_1 = \langle init_{P_1}, init_{P_2} \rangle$, and

—$\alpha_{i+1} = \langle s_{i+1}, t_{i+1} \rangle$ iff $\alpha_i = \langle s_i, t_i \rangle$ and $(s_i : m_i \rightarrow s_{i+1}) \in Transitions(P_1)$, $(t_i : m_i \rightarrow t_{i+1}) \in Transitions(P_2)$, and $Polarity(P_1, m_i) \neq Polarity(P_2, m_i)$.

By definition, $Collabs(P_1, P_2) = \{\ \alpha : \alpha$ is a collaboration history for P_1 and $P_2\ \}$. Note that every prefix of a collaboration history is a collaboration history and that $\langle init_{P_1}, init_{P_2} \rangle \in Collabs(P_1, P_2)$ so that $Collabs(P_1, P_2)$ is never empty. $Collabs(P_1, P_2)$ give all the traces that can possibly occur when P_1 and P_2 collaborate.

Protocols P_1 and P_2 have no *unspecified receptions* [Brand and Zafiropulo 1983] iff for all $\alpha \in Collabs(P_1, P_2)$, $\alpha = \alpha_1 \rightarrow_{m_1} \cdots \alpha_n$, where $\alpha_n = \langle s_n, t_n \rangle$, the following two conditions hold:

—if $s = s_n$, $(s : m \rightarrow s') \in Transitions(P_1)$, and $Polarity(P_1, m) =$ -, then there exists $\alpha' = \alpha_1 \rightarrow_{m_1} \cdots \alpha_n \rightarrow_m \alpha_{n+1} \in Collabs(P_1, P_2)$ where $\alpha_{n+1} = \langle s', t' \rangle$ for some $t' \in States(P_2)$; and

—if $t = t_n$, $(t : m \rightarrow t') \in Transitions(P_2)$ and $Polarity(P_2, m) =$ -, then there exists $\alpha' = \alpha_1 \rightarrow_{m_1} \cdots \alpha_n \rightarrow_m \alpha_{n+1} \in Collabs(P_1, P_2)$ where $\alpha_{n+1} = \langle s', t' \rangle$ for some $s' \in States(P_1)$.

That is, P_1 and P_2 have no unspecified receptions iff, whenever a collaboration α can reach the point where P_1 (P_2) is in a state where it can send a message m, its mate will be in a state where it can receive that message, and hence there exists some collaboration history in which that message is exchanged at that point.

Protocols P_1 and P_2 are *deadlock free* [Brand and Zafiropulo 1983] iff for all finite sequences $\alpha \in Collabs(P_1, P_2)$, $\alpha = \alpha_1 \rightarrow_{m_1} \cdots \alpha_{n-1} \rightarrow_{m_{n-1}} \alpha_n$, where $\alpha_n = \langle s_n, t_n \rangle$, then either

—s_n and t_n are final states of P_1 and P_2, respectively, or

—there exists $\alpha' \in Collabs(P_1, P_2)$ such that α is a *strict* prefix of α'.

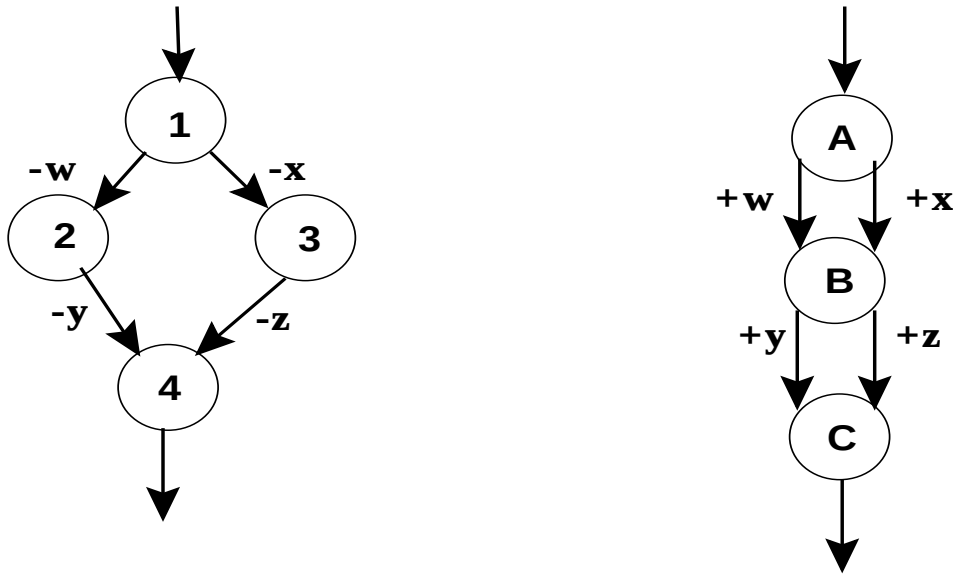

Fig. 1. Compatible protocols that accept different languages.

That is, P_1 and P_2 are deadlock free iff the collaboration α ends with both protocols in final states, or the collaboration can continue.[1]

Protocols P_1 and P_2 are *compatible* iff they have no unspecified receptions, and are deadlock free.

Our definition of protocol compatibility requires that when one party can send a message m, then the other party must be willing to receive that message. However, the protocols are compatible even when one party can receive a message m, yet the other party cannot send that message. This asymmetry reflects traditional programming environments, where the sender protocol decides what it wants to send independent of the recipient, and where the recipient has no control over what the sender sends. This differs from CSP, where the $\Box$ operator (called "alternative" or "deterministic choice") can be used to force a sender to send what the receiver wants to receive. Most of the results of this article can be rephrased to accommodate the CSP semantics.

It is interesting to note that if we look at P_1 and P_2 as language automata, then they may not accept the same language even though they are compatible in the sense described here. Figure 1 shows two such compatible protocols.

Let $s \in States(P_1)$ and $t \in States(P_2)$. By definition, $s \sim t$ iff there exists a collaboration history $\alpha = \cdots \alpha_i \cdots \in Collabs(P_1, P_2)$ where $\alpha_i = \langle s, t \rangle$. Since the relation $\sim$ captures exactly those states $s, t \in States(P_1) \times States(P_2)$ that can appear together in a collaboration state of a collaboration history in $Collabs(P_1, P_2)$,

[1]Note that our definition is based upon our previously stated assumption that a component will not indefinitely remain in a state that contains a send transition. Our definition only indicates lack of deadlock between these two components; it does not guarantee lack of deadlock within an entire system.

we can apply the criteria for compatibility given above using this relation. That is, we can reformulate the definitions of unspecified receptions and deadlock using only the $\sim$ relation. We now give an algorithm for computing this relation.

Let $EQUIV(P_1, P_2) \subseteq States(P_1) \times States(P_2)$ be the smallest set such that

—$\langle init_{P_1}, init_{P_2} \rangle$ is in $EQUIV(P_1, P_2)$; and
—if $\langle s, t \rangle$ is in $EQUIV(P_1, P_2)$, $(s : m \rightarrow s') \in Transitions(P_1)$, $(t : m \rightarrow t') \in Transitions(P_2)$, and $Polarity(P_1, m) \neq Polarity(P_2, m)$, then $\langle s', t' \rangle$ is in $EQUIV(P_1, P_2)$.

The set $EQUIV(P_1, P_2)$ is obviously finite and can be computed by a simple algorithm.

LEMMA 2.3.1. $\langle s, t \rangle \in EQUIV(P_1, P_2)$ *iff* $s \sim t$.

PROOF. For all s and t such that $s \sim t$, let $len(s \sim t) = i$ iff (1) there exists $\alpha = \alpha_1 \rightarrow_{m_1} \alpha_2 \cdots \alpha_i \rightarrow_{m_i} \cdots \in Collabs(P_1, P_2)$ where $\alpha_i = \langle s, t \rangle$ and (2) if there exists $\beta = \beta_1 \rightarrow_{r_1} \beta_2 \cdots \beta_j \rightarrow_{r_j} \cdots \in Collabs(P_1, P_2)$ with $\beta_j = \langle s, t \rangle$, then $j \geq i$. It is easy to show by induction on $len(s \sim t)$ that $s \sim t$ iff $\langle s, t \rangle \in EQUIV(P_1, P_2)$. □

We therefore have shown an algorithm for computing whether two protocols are compatible: first compute the relation $\sim$, and then use that relation to test for unspecified receptions and deadlock. This proves the following theorem:

THEOREM 2.3.2. *There exists an algorithm for checking protocol compatibility.*

As previously mentioned, one of our main reasons for adopting a synchronous semantics is that it becomes very easy to reason about protocols. In particular, for any protocol P, let $\overline{P}$ be the protocol derived from P by reversing the direction of all messages, that is, by making each receiving message a sending message and vice versa. P and $\overline{P}$ are always protocol compatible. This is not true if one adopts the asynchronous semantics.

2.4 Subprotocols

Just as *type-compatible* components are guaranteed to work together free of type errors, *protocol-compatible* components are guaranteed to work together free of protocol errors (messages arriving out of sequence or deadlock). We can extend the comparison of protocols to type systems by defining the notion of *subprotocols*, analogous to subtypes.

Informally, P is a *subprotocol* of P' iff the initial state of P is the "same" as the initial state of P'; every final state (a state without any outgoing transitions) in P' is a final state in P; and P can be obtained from P' by (1) adding a set S of new states to P, (2) adding a set of new receive transitions to P, and (3) adding a set of new send transitions to P such that each new send transition emanates from a *new* state of P (a state in S).

Formally, P is a subprotocol of P' iff there exists an injective[2] function $M : States(P') \rightarrow States(P)$ such that

[2]The fact that M is an injection (i.e., that $M(s'_1) = M(S'_2) \Rightarrow s'_1 = s'_2$) dictates that each subprotocol have a substrucuture isomorphic to its superprotocol. This constraint can be removed,

(1) $M(init_{P'}) = init_P$;
(2) s' is a final state of $P' \Rightarrow M(s')$ is a final state of P;
(3) $(s'_1 : m \rightarrow s'_2) \in Transitions(P') \Rightarrow (M(s'_1) : m \rightarrow M(s'_2)) \in Transitions(P)$; and
(4) $\forall (s_1 : m \rightarrow s_2) \in Transitions(P)$, if $M^{-1}(s_1)$ and $M^{-1}(s_2)$ are defined, but $(M^{-1}(s_1) : m \rightarrow M^{-1}(s_2)) \notin Transitions(P')$, or $M^{-1}(s_1)$ is defined, but $M^{-1}(s_2)$ is not defined, then $Polarity(m, P) = +$.

The following two lemmas show interesting properties of subprotocols.

LEMMA 2.4.1. *If P_1 is a subprotocol of P and P is compatible with P_2, then P_1 is compatible with P_2.*

PROOF. The intuition behind this proof is based upon the fact that the subprotocol P_1 can only start to deviate from the behavior that the superprotocol P exhibits once it receives a message that P was not prepared to receive. But since P_2 is compatible with P, P_2 will never send such a message in its collaboration with P. Hence it will also never send such a message in its collaboration with P_1. Therefore P and P_1 will exhibit exactly the same behavior in their collaboration with P_2. A formal proof follows.

For any $s \in States(P_1)$, let $N : States(P_1) \rightarrow States(P)$ such that $N(s) = s'$ if $M(s') = s$; otherwise $N(s) = \perp$. Let $\langle s, t \rangle$ be a collaboration state for P_1 and P_2. Then $N(\langle s, t \rangle) = \langle N(s), t \rangle$. Similarly we extend the mapping $N : Collabs(P_1, P_2) \rightarrow Collabs(P, P_2)$ such that for any $\alpha \in Collabs(P_1, P_2)$ such that $\alpha = \alpha_1 \rightarrow_{m_1} \cdots \alpha_n \rightarrow_{m_n} \cdots$, $N(\alpha) = N(\alpha_1) \rightarrow_{m_1} \cdots N(\alpha_n) \rightarrow_{m_n} \cdots$. To prove this lemma, we first show that $Collabs(P, P_2) = \{N(\alpha) | \alpha \in Collabs(P_1, P_2)\}$.

Certainly $\alpha' \in Collabs(P, P_2) \Rightarrow \exists \alpha \in Collabs(P_1, P_2)$ such that $N(\alpha) = \alpha'$, since every transition in P has a corresponding transition in P_1. We now show that the converse is also true. Assume that this is false and that there exists $\alpha \in Collabs(P_1, P_2)$ such that $N(\alpha) \notin Collabs(P, P_2)$. Let $\alpha' = \alpha_1 \rightarrow_{m_1} \alpha_2 \rightarrow_{m_2} \cdots \alpha_n$ be the longest prefix of α such that $N(\alpha') \in Collabs(P, P_2)$. (Such a prefix always exists, as $N(\alpha_1) = N(\langle init_{P_1}, init_{P_2} \rangle) = \langle init_P, init_{P_2} \rangle \in Collabs(P, P_2)$.) α must be of the form $\alpha_1 \rightarrow_{m_1} \alpha_2 \rightarrow_{m_2} \cdots \alpha_n \rightarrow_{m_n} \alpha_{n+1} \cdots$, where $\alpha_n \rightarrow_{m_n} \alpha_{n+1}$ is of the form $\langle s_n, t_n \rangle \rightarrow_{m_n} \langle s_{n+1}, t_{n+1} \rangle$. It must be that $(N(s_n) : m_n \rightarrow N(s_{n+1})) \notin Transitions(P)$; otherwise we could concatenate $N(\alpha_n) \rightarrow_{m_n} N(\alpha_{n+1})$ to the end of $N(\alpha')$, thereby obtaining a longer prefix α'' such that $N(\alpha'') \in Collabs(P, P_2)$. Since $(s_n : m_n \rightarrow s_{n+1}) \in Transitions(P_1)$, but $(N(s_n) : m_n \rightarrow N(s_{n+1})) \notin Transitions(P)$, it follows from the definition of subprotocols that m_n must be a message sent from P_2. But then $N(\alpha_n) = \langle s, t \rangle$ is a state of $N(\alpha')$ where P_2 can send the message m_n from state t_n, but P cannot receive that message. Therefore P is not compatible with P_2, contradicting our assumption.

Hence we have established a 1-1 mapping between collaborations in $Collabs(P_1, P_2)$ and those in $Collabs(P, P_2)$. One can now easily show that no collaboration history in $Collabs(P_1, P_2)$ can have an unspecified reception or deadlock, by comparing it to the corresponding collaboration history in $Collabs(P, P_2)$ and using the definition of subprotocols. □

---

thereby allowing a single state of the subprotocol to represent multiple states of the superprotocol. For simplicity, here we only allow injective functions.

LEMMA 2.4.2. *If P_1 is a subprotocol of P and P_2 is a subprotocol of $\overline{P}$, then P_1 and P_2 are compatible.*

PROOF. This lemma follows by applying Lemma 2.4.1 twice; first apply the lemma to show that P_1 is compatible with $\overline{P}$ and then to show that P_2 is compatible with P_1. □

The converse of this lemma is not true; that is, P_1 and P_2 may be compatible, even though there do not exist superprotocols A and B of P_1 and P_2, respectively, with $A = \overline{B}$. The protocols of Figure 1 illustrate such a case.

2.5 Implementation of Protocols

Protocols serve as a very useful abstraction of how components interact. There are several ways in which we can map this abstraction onto programmatic APIs that programmers can use and several ways to implement this abstraction. We now examine a few of these.

Asynchronous or Synchronous Message Passing. The component sending a message may view the send as (a) an asynchronous message, (b) a blocking send until message delivery, or (c) a blocking call until completion of a subsequent receive — i.e., a synchronous call with return. In (a) the sender immediately continues computation after sending the message. Many languages and environments support this semantic; in CORBA-compliant object systems, for instance, this can be accomplished using the *oneway* IDL keyword [Object Management Group 1995]. In (c) the sender sends a single message and blocks until the receiver finishes computation on that message. It is easy to implement this semantics using (possibly remote) procedure calls. Part (b) defines an intermediate semantics where the sender blocks until the receiver receives the message but not until the computation is finished.

In all of these implementations a single message is exchanged. If there exists a state in which one party P may send a message m and in which the subsequent state requires the mate to return message r, then one may implement that pair of transitions as a single procedure call in which P sends the parameters of m and receives back the return parameters of r.

Immediate Invocation or Queued Messages. Messages received by a component may (a) be queued until explicitly requested by the receiving component, (b) immediately invoke method handlers in the receiving component, or (c) serially invoke method handlers in the receiving component.

To implement (a), one needs an environment that makes queues explicit to the programmer. Some programming languages have built-in support for this abstraction [Arjomandi et al. 1995; Auerbach et al. 1994; Strom et al. 1991]. Alternatively, most window systems provide a queue abstraction and an event notification mechanism to inform the application of outstanding messages that need to be handled. Many RPC and distributed object systems support (b) and (c). For instance, IBM's SOM [IBM 1993] implementation of the CORBA standard provides a default *Object Server*. This server can be specified to be either single- or multithreaded. In the former case the server queues up incoming calls and dispatches

them serially. In the latter case the server starts a new thread to handle each incoming message.

Passive or Active Components. Components may be active, having autonomous threads of control, or passive, in which case threads may only exist when responding to external messages initiated from another component. Passive components are easiest to implement for protocols that consist of pairs of transitions, where the first transition is the receipt of a message, and the following transition sends backs the results of the message. In this case the component can be implemented by spawning a thread when a message is received. This thread can then handle the message and return the results to the caller. Protocols containing states in which both parties can simultaneously send usually necessitate active components.

Pessimistic or Optimistic Arbitrators. Under our formulation of the synchronous semantics, we imposed the condition that the finite-state machines describing the protocols of the two collaborating components advance atomically. This assumption made it much easier to reason about protocols and to formulate conditions for compatibility.

For implementation purposes, however, we would like to loosen this restriction. Specifically, we would like to be guaranteed that if P_1 and P_2 are compatible under the synchronous semantics, then we can implement the collaboration without requiring atomicity of send/receive transitions, but still (1) guaranteeing freedom from unspecified receptions and deadlock and (2) preserving the fact that both components agree on the *execution trace*; they agree on the total order on messages sent and received between them.

If the compatible protocols of two components contain no mixed states, then we can use any simple message-passing system to implement the message exchange, and we will still be guaranteed that the preceding two properties will be satisfied. This is not the case if the protocols contain mixed states, in which case we require some additional lower-level synchronization. To see why this is so, consider the Auctioneer protocol in Example 2.2.2 above. Suppose component C_1 implements protocol *Auctioneer*; C_2 implements $\overline{Auctioneer}$; and both C_1 and C_2 are in state B. Then suppose C_1 sends an `update` message and advances to state C. Before this message is received by C_2, C_2 sends a `bid` message. However, C_1 is now waiting for an `updateAck` message, and when the `bid` message arrives, we have an unspecified reception protocol error. Notice that the reason this occurs is that state B is a mixed state. Mixed states generate race conditions. Without some form of synchronization between components at mixed states, the two components may not agree on the execution trace and may therefore exhibit unexpected receptions or deadlock, even though their protocols are compatible under the synchronous semantics.

To synchronize protocols containing mixed states, we introduce a run-time *arbitrator* between such protocols. The arbitrator forces both partners to agree who is to send and who is to receive whenever either side may potentially send. The effect of the arbitrator is to force the two protocols to advance consistently so that if one component receives a message m_2 after sending a message m_1, the mate must have received message m_1 before sending m_2— they agree on the execution trace. In the auction example, without the arbitrator, the bidder may receive an update message

and not be able to determine whether the message was sent before or after its bid was received by the auctioneer. With arbitration such confusion is avoided.[3]

Using an arbitrator, we are guaranteed that properties (1) and (2) given above will be always satisfied. In order for this scheme to work, we need to perturb the component protocols so that they interact with the arbitrator via special *synchronization* messages.[4] In the rest of this section, we elaborate on two possible implementation schemes for arbitrators that differ in how the component interacts with the arbitrator when it is in a mixed state.

One approach, which we call the *optimistic approach*, allows either party to optimistically assume that it will win the race condition. This approach always allows a component C_1 to send a message to C_2 from a mixed state. The arbitrator will receive this message and will decide either to forward this message to C_2 or to reject the message. The arbitrator will only reject the message if it has already received a message from C_2 and declared it the winner of the race condition; that is, the arbitrator has already decided that C_1 must receive a message at this mixed state. The arbitrator will then send C_1 a reject message, telling it to "roll back" its computation to the time when it was in the mixed state, and will forward C_2's message to it. Figure 2 shows how one would perturb a protocol containing a mixed state using the optimistic approach. We should note that there are variations on this scheme that can be used to minimize the number of messages sent. For example, instead of the arbitrator explicitly sending a reject message to C_1, it can just forward C_2's message to C_1. For most protocols, C_1 will then interpret the receipt of C_2's message as an implicit rejection of its message. There are also details we are omitting concerning where the reject message should be inserted into the original protocol and what to do when multiple messages are sent optimistically. This scheme is called *optimistic* because it does not require any extra synchronization messages to be exchanged with the arbitrator when only one party tries to send from a mixed state and no race conditions arise. However, when more than one party tries to send from a mixed state, extra rollback computation needs to be performed, and extra synchronization messages need to be exchanged.

An alternative scheme never requires any rollback of computation but always entails some extra synchronization messages. In this *pessimistic approach*, a component always asks the arbitrator for permission to send a message from a mixed state. The arbitrator will either grant permission, in which case the component can (and must) send the message, or it will reject the request. The arbitrator will only reject the request if it has already received a request from the other component to send from this state and has granted permission to that component. Figure 3 shows how one would perturb a protocol containing a mixed state using the pessimistic

[3]The requirement that both applications agree on the sequence of the messages sent and received can be too restrictive in some situations. We have devised a way of generalizing our protocol language to allow one to specify that some messages can be sent in parallel by the two components. The basic idea is to introduce *parallel* and *join* nodes into the finite-state machine in which one can specify "subprotocols" that can proceed in parallel. We do not discuss these generalizations further in this article.

[4]We are assuming the use of conventional languages such as C++, which lack a construct for issuing a conditional method call. In a language like CSP, the arbitration can be performed completely transparently.

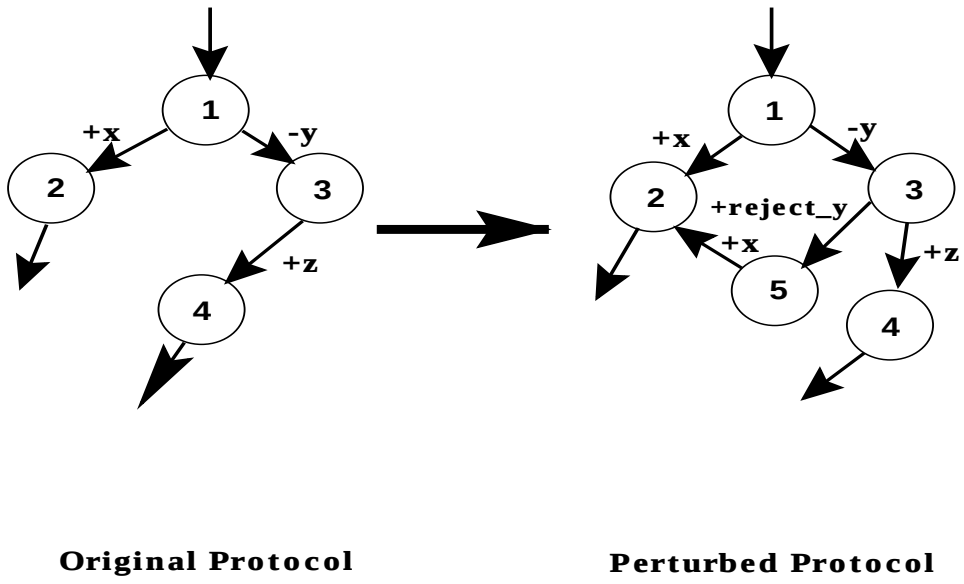

Fig. 2. Perturbing a protocol using the optimistic method.

approach.

We must note that both schemes rely on the fact that the arbitrator, when receiving a request from a component to send from a mixed state, can tell if that component has already received the messages the arbitrator has forwarded to it from its mate. This is not always the case. Consider, for instance, the protocol of Figure 4. In this case the component can repeatedly either send an x message or receive a y message. The semantics of collaborations, as mentioned earlier in the article, is that each party must agree on the *execution trace*. This means that each party must agree on the order in which the x and y messages were sent and received.

Now consider what would happen if we perturbed the protocol of Figure 4 to a pessimistic one. Say that C_1 asked permission from the arbitrator to send an x message and that this message was forwarded to C_2. Then C_2 asks permission from the arbitrator to send a y message. The arbitrator does not know if this latter request is coming before C_2 has received the last x message, in which case it must reject C_2's request, or whether it is coming after C_2 has received this message, in which case it can grant C_2's request. For either the pessimistic or optimistic schemes given above to work, it must be that the next message that can be sent from a component after receiving a message at a mixed state is different than any message it can send from that mixed state. If this criterion fails to hold, then a more sophisticated scheme must be used (requiring additional information to accompany synchronization messages). Details are beyond the scope of this article.

Putting It All Together. In summary, we have defined a system that supports the following steps:

(1) One first specifies the abstract protocols of interaction for a set of components.

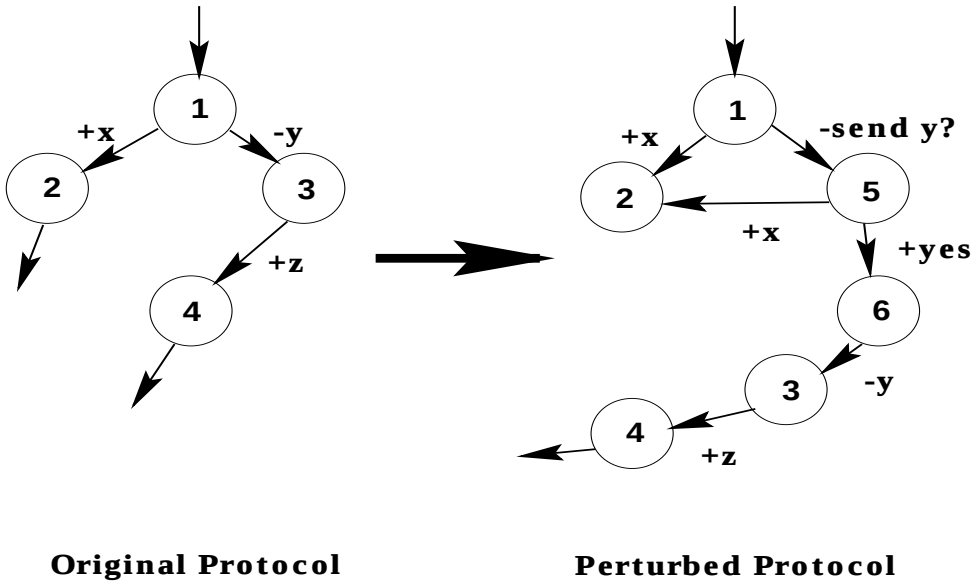

Fig. 3. Perturbing a protocol using the pessimistic method.

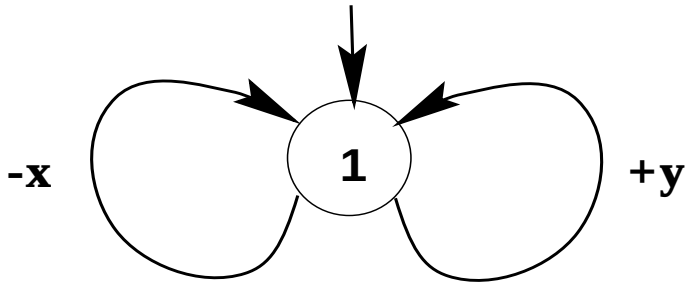

Fig. 4. A protocol requiring a more sophisticated perturbation.

(2) Next one defines, via binding pragmas, how to realize these protocols. These pragmas offer a wide variety of choices on how to reflect the abstract protocols to the programmer. Each component may want to make different choices on these bindings. A tool would automatically emit the necessary code to facilitate these bindings.

(3) A tool would also emit arbitrator code to facilitate the implementation of the synchronous semantics. Once again, how these semantics are reflected to the programmer is guided by synchronization pragmas.

The main goal of protocols is to facilitate the composition of components, even when these components have been developed in isolation from one another. Using the approach just outlined, it should be possible under many conditions to compose components that have compatible protocols, even if they realize those protocols via different binding pragmas. This approach can help alleviate some of the practical problems of component composition (see Garlan et al. [1995]).

3. ADAPTORS

When one component provides a service that another component requires, it is often not possible to bind the two components together if they were not programmed to compatible collaboration specifications. An *adaptor* is a piece of code that sits between two components and compensates for the differences between their interfaces.

Example 3.1. The following shows a Bidder collaboration specification. When the auction begins (or when the bidder attaches to the auction in progress), the bidder receives an `auctionBegin` message from the auctioneer containing information about the current item being auctioned. The bidder then enters state 2, where it can request participation in the auction by sending a `requestToBid` message and enters state 3. From this state the bidder either receives a `cannotBid` or a `canBid` message from the auctioneer. In the latter case, the bidder is supplied an id to use on subsequent bids. Once granted permission to participate in the auction, the bidder enters state 4, where it will either receive `newHighBid` messages informing it about new high bids for the item or a `gavel` message, indicating that the auction for this item is over. When in state 4 the bidder may also place a bid by sending a `newBid` message, in which case it enters state 6. The auctioneer responds to the bidder's bid by either sending a `bidNotOk` or a `bidOk` message. In the former case, the bidder moves back to state 4; in the latter case it moves to state 7, representing the fact that this bidder now owns the high bid for the item. From this state the auctioneer can either (1) inform the bidder of subsequent higher bids for the item received from competing bidders by way of a `newHighBid` message or (2) signal that the auction is over and that this bidder has bought the item. Each `newHighBid` message is expected to be acknowledged by the bidder with a `highBidAck` message.

```
Collaboration Bidder {
  Receive Messages {
    auctionBegin(auctionInfo:string);
    canBid(bidId:id);
    cannotBid(why:string);
```

```
    newHighBid(price:real);
    bidOk();
    bidNotOk();
    gavel();
  };
  Send Messages {
    requestToBid(name:string,bidItem:string);
    newBid(bidId:id, myBid:real);
    highBidAck();
  };
  Protocol {
    States {1 (init), 2, 3, 4, 5, 6, 7};
    Transitions {
      1: +auctionBegin -> 2;
      2: -requestToBid -> 3;
      3: +cannotBid    -> 1;
      3: +canBid       -> 4;
      4: +newHighBid   -> 5;
      4: +gavel        -> 1;
      4: -newBid       -> 6;
      5: -highBidAck   -> 4;
      6: +bidNotOk     -> 4;
      6: +bidOk        -> 7;
      7: +newHighBid   -> 5;
      7: +gavel        -> 1;
    };
  };
};
```

This bidder specification is not compatible with the Auctioneer specification given in Example 2.2.2. There are several minor ways in which these specifications differ, such as different message and parameter names and different parameter orders. Additionally some parameters are missing. Even more substantially, this protocol has seven states, as opposed to the five states of `Auctioneer`. This is because this protocol has a "two-phase" property: instead of just bidding on the current auction item, this protocol demands that this bidder first "authenticate" itself and ask permission to bid before actually bidding on the current item being auctioned. Hence bidder states 1,4,5,6,7 correspond roughly to auctioneer states A,B,C,D,E, but bidder states 2 and 3 have no counterpart. This introduces another incompatibility; the `bid` message expected by the auctioneer has an `itemBiddingOn` parameter, whereas the corresponding message sent by the bidder, `newBid`, lacks this parameter. However, an equivalent parameter `bidItem` is supplied by the `requestToBid` message, which is only issued once per item being auctioned. As we will see, we will need to produce an adaptor that will synthesize the parameters for the `bid` message from both the `newBid` message and the `requestToBid` message and which will *remember* the `bidItem` parameter of the `requestToBid` message for subsequent reuse. Similarly, the `bidderId` supplied by the auctioneer at the start of the col-

laboration is not expected by the bidder until it receives permission to bid. The adaptor will need to store this parameter until it can be forwarded to the bidder.

3.1 Adaptor Specifications

We model an adaptor as a finite-state machine that has interfaces to the two components that want to collaborate. All messages exchanged by the components will go through the adaptor. The adaptor's behavior is governed by its transition rules. If the adaptor is in a state with a send transition, the adaptor may send a message to the appropriate component and enter the target state. If the adaptor is in a state with a receive transition, the adaptor may wait for a message to arrive and then enter the target state. Additionally, we allow an *action* to be performed by the adaptor whenever a transition is taken. The actions we allow are simple enough to easily analyze interesting properties of adaptors, while being generic enough to capture most adaptor behaviors.

Let $\mathcal{A}$ be an adaptor for two components C_1 and C_2 supporting protocols P_1 and P_2, respectively. $\mathcal{A}$ will support a protocol compatible with P_1 in its collaboration with C_1 and a protocol compatible with P_2 in its collaboration with C_2. P_1 and P_2 need not be protocol-compatible; they may not even support the same set of messages!

An adaptor $\mathcal{A}$ is specified by a tuple $\langle States_A, Cells_A, Rules_A \rangle$. $States_A$ is a finite set of states. $Cells_A$ consists of a finite set of *typed* memory cells. It will contain exactly one memory cell, $Cells.parm$, for each parameter *parm* that can be received by the adaptor $\mathcal{A}$. For each state in $States_A$, a particular subset of cells from $Cells_A$ will be *valid*, that is, will hold a value saved from a previously received message. $Rules_A$ consists of a set of state transitions augmented with *memory actions*. A state transition involves either sending a message to or receiving a message from one of the components C_1 or C_2. A rule has one of the following two forms:

```
<s1>: + <message> from <component_name> -> <s2>
  [ , <save_actions> ]
  [ , <invalidate_actions> ] ;

<s1>: -  <message> to <component_name> -> <s2>
  [, <synthesis_actions> ]
  [, <invalidate_actions> ] ;
```

The semantics of the first form of a rule is as follows: when in state `<s1>` the adaptor can receive a message of type `<message>` from `<component_name>` (either C_1 or C_2) and will then advance to state `<s2>`. Additionally the rule can specify the memory actions `<save_actions>` and `<invalidate_actions>` to be taken. Each `<save_action>` is of the form `write(parm)` indicating that the `parm` parameter of the message should be stored in cell $Cells.parm$. `write(parm1,parm2,...)` is a shorthand for `write(parm1), write(parm2),...`. A written memory cell becomes valid, and we say that the rule *writes* $Cells.parm$. We explain the `<invalidate_actions>` clause below.

The semantics of the second form of a rule is as follows: when in state `<s1>` the adaptor can send a message of type `<message>` to `<component_name>` (either

C_1 or C_2) and will then advance to state `<s2>`. Additionally, the rule *must* specify where the value of *each* parameter of the message comes from. The actions `<synthesis_actions>` serve this purpose. Each `<synthesis_action>` is of the form `parm = x` or of the form `parm = f(x1,x2,...)`, where `parm` is a parameter of the message, and each `x,x1,x2,...` is either a constant or a memory cell. In the former case the parameter `parm` is set equal to the value of `x`, and in the latter case `parm` is set equal to the value of the function `f` applied to `x1,x2,...`. We say that the rule *reads* `x` (`x1,x2,...,` respectively).

In either rule, one may optionally include an `<invalidate_actions>` clause of the form

```
invalidate(x1,x2,...),
```

where `x1,x2,...` are memory cells. This indicates that the value of these memory cells is to be no longer retained — these cells cease to be valid. This allows one to state rules such as "after receiving a `restart` message, invalidate all previously valid memory cells," or "after using $Cells.p$ to synthesize parameters for message `m`, invalidate $Cells.p$."

3.2 An Example Adaptor

Example 3.2.1. Figure 5 shows an adaptor specification for Auctioneer and Bidder components supporting the protocols of Examples 2.2.2 and 3.1, respectively. For brevity we have left out all transitions relating to the receipt of the `itemSold` message. Each state of the adaptor is represented by a tuple `<x,y,mem>`, where `x` is a state of the Auctioneer protocol, `y` is a state of the Bidder protocol, and `mem` lists those memory cells that are valid in this adaptor state.

3.3 Adaptor Compatibility

In the rest of this section we extend the terminology and proofs of protocol compatibility given in Section 2.3 to include collaborations involving an adaptor.

We write $(u : m \rightarrow u') \in Transitions(\mathcal{A}, P_i)$ $(i = \{1, 2\})$ to mean that the given transition is a transition in adaptor $\mathcal{A}$ used to communicate with the component implementing protocol P_i.

A *collaboration state* for protocols P_1 and P_2 using an adaptor $\mathcal{A}$ is a tuple $\langle s, t, u\rangle$, where $s \in States(P_1)$, $t \in States(P_2)$, and $u \in States(\mathcal{A})$. A *collaboration history* for protocols P_1 and P_2 using adaptor $\mathcal{A}$ is a possibly infinite sequence of the form: $\alpha_1 \rightarrow_{m_1} \alpha_2 \rightarrow_{m_2} \cdots$, where

—each α_i is a collaboration state for P_1 and P_2 using the adaptor $\mathcal{A}$,
—$\alpha_1 = \langle init_{P_1}, init_{P_2}, init_{\mathcal{A}}\rangle$,
—$\alpha_{i+1} = \langle s_{i+1}, t_{i+1}, u_{i+1}\rangle$ iff $\alpha_i = \langle s_i, t_i, u_i\rangle$, and either

(1) $(s_i : m_i \rightarrow s_{i+1}) \in Transitions(P_1)$, $(u_i : m_i \rightarrow u_{i+1}) \in Transitions(\mathcal{A}, P_1)$, $Polarity(P_1, m_i) \neq Polarity(\mathcal{A}, m_i)$, and $t_{i+1} = t_i$ or
(2) $(t_i : m_i \rightarrow t_{i+1}) \in Transitions(P_2)$, $(u_i : m_i \rightarrow u_{i+1}) \in Transitions(\mathcal{A}, P_2)$, $Polarity(P_2, m_i) \neq Polarity(\mathcal{A}, m_i)$, and $s_{i+1} = s_i$.

By definition, $Collabs(P_1, P_2, \mathcal{A}) = \{\ \alpha : \alpha$ is a collaboration history for P_1 and P_2 using $\mathcal{A}\}$.

```
/* LEGEND
   M0 = {}
   M1 = {itemDescr,bidderId}
   M2 = {bidderId}
   M3 = {bidItem,bidderId}
   M4 = {bidItem}
   M5 = {bidItem,highBid}
   M6 = {bidItem,bidId,myBid}
   M7 = {bidItem,highBid,bidderId}
   M8 = {highBid,bidderId}  */
<A,1,M0>: +newItem from auctioneer -> <B,1,M1>,
   write(itemDescr, bidderId);
<B,1,M1>: -auctionBegin to bidder   -> <B,2,M2>,
   auctionInfo = Cells.itemDescr,
   invalidate(itemDescr);
<B,2,M2>: +requestToBid from bidder -> <B,3,M3>,
   write(bidItem);
<B,2,M2>: +update from auctioneer   -> <C,2,M8>,
   write(highBid);
<B,3,M3>: -canBid to bidder         -> <B,4,M4>,
   bidId = Cells.bidderId,
   invalidate(bidderId);
<B,4,M4>: +update from auctioneer   -> <C,4,M5>,
   write(highBid);
<B,4,M4>: +newBid from bidder       -> <B,6,M6>,
   write(bidId, myBid);
<C,4,M5>: -newHighBid to bidder     -> <C,5,M4>,
   price = Cells.highBid,
   invalidate(Cells.highBid);
<C,5,M4>: +highBidAck from bidder   -> <C,4,M4>;
<C,4,M4>: -updateAck to auctioneer-> <B,4,M4>;
<B,6,M6>: -bid to auctioneer        -> <D,6,M4>,
   bidderId = Cells.bidId,
   itemBiddingOn = Cells.bidItem,
   amount = Cells.myBid,
   invalidate(Cells.bidId,Cells.myBid);
<D,6,M4>: +rejectBid from auctioneer-> <B,6,M4>;
<D,6,M4>: +acceptBid from auctioneer-> <E,6,M4>;
<B,6,M4>: -bidNotOk to bidder     -> <B,4,M4>;
<E,6,M4>: -bidOk to bidder          -> <E,7,M4>;
<E,7,M4>: +update from auctioneer   -> <C,4,M5>,
   write(highBid);
<C,2,M8>: -updateAck to auctioneer  -> <B,2,M8>;
<B,2,M8>: +update from auctioneer -> <C,2,M8>,
   write(highBid);
<B,2,M8>:+requestToBid from bidder-> <B,3,M7>,
   write(bidItem);
<B,3,M7>: -canBid to bidder       -> <B,4,M5>,
   bidId = Cells.bidderId,
   invalidate(bidderId);
<B,4,M5>: -newHighBid to bidder   -> <B,5,M4>,
   price = Cells.highBid;
   invalidate(Cells.highBid);
<B,5,M4>: +highBidAck from bidder -> <B,4,M4>;
```

Fig. 5. An adaptor for the Auctioneer and Bidder protocols.

We now define deadlock and unspecified receptions in a manner analogous to Section 2.3. Protocols P_1 and P_2 using adaptor $\mathcal{A}$ have no *unspecified receptions* iff for all $\alpha \in Collabs(P_1, P_2, \mathcal{A})$, $\alpha = \alpha_1 \rightarrow_{m_1} \cdots \alpha_n$, where $\alpha_n = \langle s_n, t_n, u_n \rangle$, the following conditions hold:

—if $s = s_n$ is *not* a mixed state in P_1, $(s : m \rightarrow s') \in Transitions(P_1)$, and $Polarity(P_1, m) =$-, then there exists $\alpha' = \alpha_1 \rightarrow_{m_1} \cdots \alpha_n \rightarrow_m \alpha_{n+1} \in Collabs(P_1, P_2, \mathcal{A})$, where $\alpha_{n+1} = \langle s', t_n, u' \rangle$ for some $u' \in States(\mathcal{A})$,

—if $t = t_n$ is *not* a mixed state in P_2, $(t : m \rightarrow t') \in Transitions(P_2)$ and $Polarity(P_2, m) =$-, then there exists $\alpha' = \alpha_1 \rightarrow_{m_1} \cdots \alpha_n \rightarrow_m \alpha_{n+1} \in Collabs(P_1, P_2, \mathcal{A})$, where $\alpha_{n+1} = \langle s_n, t', u' \rangle$ for some $u' \in States(\mathcal{A})$,

—if $u = u_n$, $(u : m \rightarrow u') \in Transitions(\mathcal{A}, P_1)$ and $Polarity(\mathcal{A}, m) =$-, then there exists $\alpha' = \alpha_1 \rightarrow_{m_1} \cdots \alpha_n \rightarrow_m \alpha_{n+1} \in Collabs(P_1, P_2, \mathcal{A})$, where $\alpha_{n+1} = \langle s', t_n, u' \rangle$ for some $s' \in States(P_1)$,

—if $u = u_n$, $(u : m \rightarrow u') \in Transitions(\mathcal{A}, P_2)$ and $Polarity(\mathcal{A}, m) =$-, then there exists $\alpha' = \alpha_1 \rightarrow_{m_1} \cdots \alpha_n \rightarrow_m \alpha_{n+1} \in Collabs(P_1, P_2, \mathcal{A})$, where $\alpha_{n+1} = \langle s_n, t', u' \rangle$ for some $t' \in States(P_2)$.

Note that we do not require the adaptor to be able to accept every message that may be sent to it from a component if that message originates in a mixed state of the component. The reason for this is to allow an adaptor to be in a state where it can receive a message m_1 from C_1 or a message m_2 from C_2, but once it receives a message from one of the components, say C_1, it advances to a state where it will no longer accept m_2 messages from C_2, as long as C_2 is in a mixed state.

This idiom is quite useful, as illustrated by our sample adaptor of Example 3.2.1. When it is in state `<B,4>` it can receive an `update` message from the auctioneer or a `newBid` message from the bidder. If the former message is received first, then the adaptor "commits" itself to forwarding this message as a `newHighBid` message to the bidder, and the adaptor will no longer accept `newBid` messages from the bidder before this `newHighBid` message is acknowledged by the bidder. Similarly, if a `newBid` message arrives first, the adaptor commits itself to forwarding this message as a `bid` message to the auctioneer and will no longer accept `update` messages before this `bid` message is replied to by the auctioneer. In this way the adaptor need not concern itself with these messages "crossing." This simplifies both protocols and implementations. These semantics have implications on the construction of arbitrators for adaptors, as we discuss at the end of Section 3.5.

Protocols P_1 and P_2 using adaptor $\mathcal{A}$ are *deadlock free* iff for all finite sequences $\alpha \in Collabs(P_1, P_2)$, $\alpha = \alpha_1 \rightarrow_{m_1} \cdots \alpha_{n-1} \rightarrow_{m_{n-1}} \alpha_n$, where $\alpha_n = \langle s_n, t_n, u_n \rangle$, then either

—s_n and t_n are final states of P_1 and P_2, respectively, or

—there exists $\alpha' \in Collabs(P_1, P_2, A)$ such that α is a *strict* prefix of α'.

An adaptor $\mathcal{A}$ is *compatible* with protocols P_1 and P_2 iff they have no unspecified receptions and are deadlock free.[5]

[5]According to our definition, if progress can always be made between the adaptor and one component, then the system is deadlock free, even if the other component has reached a deadlock state

THEOREM 3.3.1. *There exists an algorithm for checking whether an adaptor $\mathcal{A}$ is compatible with protocols P_1 and P_2.*

PROOF. The proof of this lemma is analogous to the proof of protocol compatibility (without an adaptor) in Section 2.3. First we define the relation $\sim \subseteq States(P_1) \times States(P_2) \times States(\mathcal{A})$ such that $\sim (s, t, u)$ iff there exists some collaboration history $\alpha = \cdots \alpha_i \cdots \in Collabs(P_1, P_2, \mathcal{A})$ where $\alpha_i = \langle s, t, u \rangle$. We then give an algorithm that computes the relation $\sim$. This algorithm is similar to the algorithm for computing $EQUIV$ in Section 2.3. Given the relation $\sim$, it is straightforward to check for compatibility. We leave the details of the proof for the reader. □

3.4 Limitations of Adaptors

There are protocols that are incompatible under the synchronous semantics of protocol collaboration, even though they would be compatible under an asynchronous semantics. Some of these incompatible protocols can be made to work together using an adaptor, but not all.

Consider, for instance, two protocols P_1 and P_2 and messages m_1, containing a parameter p_{m_1}, and m_2, containing a parameter p_{m_2}. Say that P_1 first sends message m_1 and then receives message m_2 and that P_2 first sends message m_2 and then receives m_1. These protocols are incompatible under the synchronous semantics (as they would not agree on the execution trace), but an adaptor can be used to make components implementing these protocols work together.

On the other hand, if P_1 first sends two m_1 messages and then receives two m_2 messages, and P_2 sends two m_2 messages and then receives two m_1 messages, the two cannot be made to work together even using an adaptor (although they would be compatible using traditional asynchronous semantics). The reason is that our formulation of adaptors only allowed the adaptor to store *one* copy of any parameter. In this case, it could not store the parameter p_{m_1} of the first m_1 message as well as the second m_1 message simultaneously. We can easily get around this problem by generalizing our definition of adaptors, allowing an adaptor to have a memory cell for each parameter of *each transition* in the protocol. This would not solve the more general case, however, where P_1 and P_2 can each send an unbounded number of messages before receiving the messages sent by the other party. Allowing this sort of functionality would require the adaptor to store an unbounded number of message parameters and would lead to undecidability results regarding adaptor properties.

3.5 Properties of Adaptors

Given protocols P_1 and P_2, and an adaptor $\mathcal{A}$, we would like to be able to ascertain whether certain properties are guaranteed for any collaboration between two components using protocols P_1 and P_2 and adaptor $\mathcal{A}$. We are interested in three kinds of properties:

where it cannot progress any further. A stronger definition would require the lack of deadlock for both components. We could equally as well have used the stronger definition.

—Memory consistency: Is the adaptor specified in such a way that it will always have received a message before trying to forward a parameter from that message?
—Parameter relationships: Does the adaptor maintain the correct relationship between the parameters of P_1 and those of P_2?
—Patterns of collaboration: Does the adaptor enforce the correct *pattern* of communication between P_1 and P_2?

It should be clear that checking these properties is useful to guarantee that the adaptor will actually function as desired; given a specification of the properties we expect an adaptor to exhibit, we can check that a particular adaptor conforms to this specification. Furthermore, in the next section we show how we can *synthesize* an adaptor from the specification.

Memory Consistency. By definition, an adaptor $\mathcal{A}$ contains a set of memory $Cells$: one cell $Cells.parm$ for each parameter *parm* that can be received by the adaptor. A cell $Cells.p$ is *valid* at a state α_k of a collaboration history $\alpha = \alpha_1 \rightarrow_{m_1} \alpha_2 \rightarrow_{m_2} \cdots \alpha_k \cdots$ iff there exists a state $\alpha_i = \langle s_i, t_i, u_i \rangle$, $i < k$, such that the transition into u_i writes $Cells.p$, and there does not exist a state $\alpha_j = \langle s_j, t_j, u_j \rangle$, $i \leq j < k$, such that the transition out of u_j invalidates $Cells.p$. A collaboration history $\alpha = \alpha_1 \rightarrow_{m_1} \alpha_2 \rightarrow_{m_2} \cdots \in Collabs(P_1, P_2, \mathcal{A})$ is *memory consistent* iff for all i and for all cells $Cells.p$, if the transition from α_i into α_{i+1} causes $\mathcal{A}$ to read cell $Cells.p$, then $Cells.p$ is valid at α_i. $\mathcal{A}$ is memory consistent iff for all $\alpha \in Collabs(P_1, P_2, \mathcal{A})$, α is memory consistent.

A proof of the following lemma is given in the Appendix.

LEMMA 3.5.1. *There exists an algorithm for checking whether or not an adaptor is memory consistent.*

By definition, an adaptor is *well formed* w.r.t. P_1 and P_2 iff it is compatible with P_1 and P_2 and is memory consistent. The well-formedness of an adaptor is the minimum that is required for the adaptor to be correct. For if it is not well formed then either there exists a mismatch in protocols between the adaptor and one of the components it is communicating with, or the adaptor may attempt at some point to forward a value to a component when it has not yet received that value from the other component, or when it has already forwarded the value and subsequentially invalidated it. The adaptor of Example 3.2.1 (Figure 5) is well formed.

THEOREM 3.5.2. *There exists an algorithm to check whether an adaptor is well formed.*

The proof of this theorem follows immediately from Theorem 3.3.1 and Lemma 3.5.1.

Parameter Relationships. Another property of interest is how parameters sent from the adaptor are related to parameters received by the adaptor. By definition, `val` $\triangleright$ `parm` means that there exists some adaptor rule that synthesizes the `parm` parameter of a message from the value `val`. That is, if there exists a `<synthesis_action>` of the form `parm = x` or of the form `parm = f(x1,x2,...)`, then `y` $\triangleright$ `parm` or `yi` $\triangleright$ `parm` ($1 \leq i$), where `y = x (yi = x)` if `x` is a constant, and `y = p (yi = p)` if `x = Cells.p`. In our example adaptor we have the property `itemDescr` $\triangleright$ `auctionInfo`, as there is a rule that forwards the `itemDescr`

parameter as the `auctionInfo` parameter. Given an adaptor specification, it is straightforward to discover all relationships of the form `X` $\triangleright$ `Y`.

Additionally, we often want to know whether a parameter received by the adaptor (and stored into a memory cell) is used just once or is used repeatedly to synthesize parameters. In our example adaptor the parameter `itemDescr` is stored into a memory cell and then used just once to define the `auctionInfo` parameter before it is invalidated. In contrast, the `bidItem` parameter, once received from the bidder in a `requestToBid` message, can be used repeatedly to define the `itemBiddingOn` parameter each time a `bid` message is sent to the auctioneer. We write `one-shot(p)` if any particular instance of the parameter `p` is used at most once to define a parameter synthesized by the adaptor. In the Appendix we also prove the following lemma.

LEMMA 3.5.3. *Given a memory-consistent adaptor, there exists an algorithm to determine whether the adaptor satisfies the property* `one-shot(p)` *for some parameter* `p`.

Patterns of Collaborations. The final class of adaptor properties we are concerned with is causality relationships between messages that the adaptor receives and messages that it sends. We use a regular-expression language to formulate these relationships. A *pattern* is a regular expression, where the alphabet is the set of messages that can either be received or sent by the adaptor. A pattern is constructed by the following grammar:

```
<pattern> ::= <message_name>
            | <pattern> + <pattern>
            | <pattern> . <pattern>
            | <pattern> *
            | ( <pattern> )
```

If a pattern is `<message_name>` it means that the adaptor receives or sends that message and then terminates (the interface specification determines whether the message is a sending or receiving message in the adaptor). If a pattern is of the form

```
<pattern1> + <pattern2>
```

it means that the adaptor sends and/or receives the message sequence specified by *either* `<pattern1>` or `<pattern2>` and then terminates. If the pattern is of the form

```
<pattern1> . <pattern2>
```

it means that the adaptor first sends and/or receives the message sequence specified by `<pattern1>` and then sends and/or receives the message sequence specified by `<pattern2>` and then terminates. Finally, if the pattern is of the form `<pattern>*` it means that the adaptor sends and/or receives the message sequence specified by `<pattern>` zero or more times. We use the notation `!(m1 + m2 + ...)` to mean that the adaptor sends or receives a message *other than* the specified messages. This is just a notational convenience, since the set of messages that can be received or sent by the adaptor is finite, and we could just as well have enumerated the set of messages it can receive or send, rather than the ones it cannot.

Here are some examples. Let `m1` be a receiving message and `m2` be a sending message. An adaptor satisfies the pattern `m1.m2` iff, for every collaboration, it first receives an `m1` message, next sends an `m2` message, and then terminates. It satisfies the pattern

```
( (!(m1 + m2))* .  m1 .  (!(m1 + m2))* . m2)* (!(m1 + m2))*
```

iff, for every collaboration, every time the adaptor receives an `m1` message it subsequently sends an `m2` message, and whenever the adaptor sends a `m2` message it must have previously received an `m1` message. Furthermore, no two `m1` messages can be received without an intervening `m2` message being sent, and no two `m2` messages can be sent without an intervening `m1` message being received. This pattern captures the fact that there is often a one-to-one relationship between messages that one protocol sends and messages that the other protocol receives. A similar pattern is `(!m0)* .m0 . ((!(m1 + m2))* .  m1 . (!(m1 + m2))* .m2)*(!(m1 + m2))*`. This pattern states that the preceding conditions on `m1` and `m2` messages only apply once the adaptor receives/sends an `m0` message.

As a pattern is a regular expression, we can build a deterministic automaton to recognize the pattern. As an adaptor executes, it sends and receives messages. At any given time during its execution, the adaptor has either *violated* the pattern, or it is at some state within the pattern automaton. When the adaptor starts execution it is at the initial state of the pattern automaton. If the adaptor sends or receives a message m, and the adaptor is at a pattern automaton state p that does not allow the sending or receipt of m, then the adaptor *violates* the pattern. Otherwise the adaptor moves to the pattern automaton state that follows p upon the receipt or sending of m. If the adaptor is at a *recognizing* pattern automaton state (i.e., a final state of the pattern automaton), then the adaptor at that state *recognizes* the pattern. Because the pattern may indicate a potentially infinite message sequence, an adaptor can fluctuate infinitely often from a state that recognizes the pattern to one that does not. For instance, if the pattern is of the form `(pattern1 . pattern2)*`, and the adaptor has sent and/or received the message sequence specified by `pattern1 . pattern2 . pattern1`, then it has not violated the pattern but is not in a state that recognizes the pattern either. If it next sends and/or receives the message sequence specified by `pattern2`, it then is in a state that recognizes the pattern.

We want to be able to statically determine if an adaptor conforms to the message sequences described by a particular pattern. By definition, an adaptor *weakly satisfies* a pattern if it is guaranteed to never violate the pattern, and if the adaptor ever terminates in a final state, it is guaranteed to be a state that recognizes the pattern. Under weak satisfaction it may be that the adaptor does not violate the pattern but will also never be in a state that recognizes the pattern. For instance, given the pattern `(m1 . (!m1)* . m2)*` it may be that the adaptor receives `m1` and then performs an infinite amount of communication without sending `m2`. In this case, as long as it does not receive another `m1` message in the meantime it weakly satisfies the pattern.

An adaptor *strongly satisfies* a pattern if it is guaranteed to never violate the pattern, and if the adaptor is ever in a state that does not recognize the pattern, it is guaranteed to eventually reach a state that does recognize the pattern. For

instance, in the pattern given above, it would require that the adaptor always eventually forward an `m2` message after receiving an `m1` message, even if the adaptor never terminates.

Strong satisfaction is a stronger notion than weak satisfaction, as it includes a notion of *progress*. However, strong satisfaction allows us to state stronger properties than we can state with our protocol language. Consider, for instance, a protocol that first receives m_1 and then sends m_2. Our protocol semantics is that after receiving m_1 it will send m_2. Hence we can prove that such a protocol strongly satisfies the pattern `m1.m2`. However, consider the protocol that first receives an unbounded number of m_1 messages and then sends an m_2 message. Our semantics does not require that the component ever send an m_2 message, and it is actually *impossible* in our current protocol language to specify "the component/adaptor will receive an unbounded number of m_1 messages and then send an m_2 message." Hence no adaptor would ever strongly satisfy the pattern `m1*.m2`. This suggests a future research direction to extend our protocol specification language with additional constructs from temporal logic.

LEMMA 3.5.4. *There exist algorithms to determine whether an adaptor weakly* (*strongly*) *satisfies a pattern.*

The proof of this lemma is given in the Appendix.

Adaptors and Arbitrators. Just as we require an arbitrator to enforce the synchronous semantics between two collaborating components, we likewise require arbitrators for collaborations between an adaptor $\mathcal{A}$ and the components C_1 and C_2 it collaborates with. Unfortunately we *cannot* use one arbitrator for the collaboration between $\mathcal{A}$ and C_1 and a separate arbitrator for the collaboration between $\mathcal{A}$ and C_2.

This is because the adaptor $\mathcal{A}$ and a component, say C_1, may be in mixed states allowing either party to send, and the adaptor may then switch to a state disallowing C_1 to send, even though no message exchange has occurred between C_1 and $\mathcal{A}$. This can only occur, however, when a message exchange occurs between $\mathcal{A}$ and its other collaborating component C_2 (see Section 3.3). If an arbitrator was only aware of messages between $\mathcal{A}$ and C_1 it would not be able to enforce this protocol.

To solve this problem we must construct a *single* arbitrator between $\mathcal{A}$ and both its collaborating components C_1 and C_2. This arbitrator will be aware of all the messages exchanged between $\mathcal{A}$ and C_1 and C_2. Alternatively, $\mathcal{A}$ can itself assume the role of arbitrator.

We can therefore follow the same methodology for collaborations between an adaptor and components as between two components: we reason about the collaboration using abstract protocols of all the parties. Given compatibility between them, we implement the synchronous semantics by constructing an arbitrator (using either optimistic or pessimistic techniques as described in Section 2.5). We augment the abstract protocols with the appropriate synchronization messages. The arbitrator for the adaptor and collaborating components constructed in this case needs to potentially be aware of all the messages exchanged between the three parties.

4. ADAPTOR SYNTHESIS

Although one can specify an adaptor using the techniques of the previous section, it can be tedious and error prone to specify all the transitions and actions. We would like a tool that takes a very concise declarative specification that relates parameters and messages in the two different collaboration specifications and either automatically synthesizes a well-formed adaptor consistent with that specification or determines that no such adaptor exists. In this section we show how this can be done.

4.1 Interface Mappings

Let $S1$ and $S2$ be the interface signatures of two collaboration specifications. An *interface mapping* between $S1$ and $S2$ consists of a set of rules, where each `<MappingRule>` is given by the following syntax:

```
<MappingRule> ::=  <parm_mapping_rule>
                 | <parm_usage_rule>
                 | <causality_rule>

<parm_mapping_rule> ::=
 [<Function>] <parm_or_const>+ -> <parm>;

<parm_usage_rule> ::= one-shot ( <parm> );

<causality_rule> ::=
   forward <msg1> as <msg2>;
 | if <msg0> then forward <msg1> as <msg2>;
 | <pattern>

<parm_or_const> ::=   <constant>
                    | <parm>
<parm> ::=
  <component_name>::<msg_name>.<parm_name>

<msg> ::= <component_name>::<msg_name>
```

Each rule is either a `<parm_mapping_rule>`, a `<parm_usage_rule>`, or a `<causality_rule>`. As described in the following, these rules relate messages and parameters in the two interfaces. Each rule in an interface mapping $\mathcal{I}$ introduces a constraint that any *correct* adaptor $\mathcal{A}$ for these collaboration specifications must obey.

The left-hand side of a `<parm_mapping_rule>` consists of an optional conversion function and a list of parameters and/or constants. (The `+` metasymbol indicates one or more instances of the preceding syntactic element.) The right-hand side of the rule consists of a single parameter. If the rule is of the form `f(x1,...,xn)->p`, then the value of `p` is obtained by evaluating `f` on its arguments. Otherwise the rule is of the form `x->p`, in which case the value of `p` is obtained directly from the value of `x`. If a parameter `cn::mn.pn` is on the left-hand side of this rule, `pn` must be the name of a parameter sent by the component `cn` in message `mn`. If the right-hand

side is of the form `cn'::mn'.pn'`, then `pn'` must be the name of a parameter to be received by the component `cn'` in message `mn'`. If an adaptor $\mathcal{A}$ is correct w.r.t. $\mathcal{I}$ then the adaptor satisfies the property `val` $\triangleright$ `parm` only if there exists a mapping rule in $\mathcal{I}$ with `val` on the left-hand side and `parm` on the right-hand side.

A `<parm_usage_rule>` is of the form `one-shot(cn::mn.pn)`. It indicates that any instance of the parameter `pn` sent by component `cn` in message `mn` should be used to define at most one parameter synthesized by the adaptor. An adaptor $\mathcal{A}$ is correct w.r.t. $\mathcal{I}$ only if the adaptor satisfies each one-shot rule given in $\mathcal{I}$.

A `<causality_rule>` is either of the form `forward cn1::mn1 as cn2::mn2` or `if cn0::mn0 then forward cn1::mn1 as cn2::mn2`, where `mn0` is a message sent by component `cn0`; `mn1` is a message sent by component `cn1`; and `mn2` is a message received by `cn2`. Alternatively, a causality can be any pattern of the type described in Section 3.5. The first two forms are just shorthand notations for frequently used patterns. The first form is a shorthand for the pattern `( (!(m1 + m2))* . m1 . (!(m1 + m2))* . m2)*(!(m1 + m2))*`. As discussed in Section 3.5, this means that the adaptor will faithfully maintain a 1-1 correspondence between `m1` messages it receives and `m2` messages it sends. The second form is similar to the first, except that it states that the correspondence need not be maintained until component `cn0` sends message `mn0`. An adaptor $\mathcal{A}$ is correct w.r.t. $\mathcal{I}$ only if, for every collaboration, the adaptor satisfies each pattern specified in a causality rule in $\mathcal{I}$. Recall that in Section 3.5, we defined both weak and strong satisfaction of patterns. We assume weak satisfaction for the reasons discussed previously. However, as stated in Lemma 3.5.4, it is also possible to check an adaptor for strong satisfaction of patterns.

An interface mapping is *complete* iff every parameter `pn` of a message `mn` that can be received by component `cn` appears on the right-hand side of some `<parm_mapping_rule>`, unless a causality rule dictates that the message `mn` is never to be sent to component `cn` (i.e., via a pattern `(!cn::mn)*`). Completeness is an essential property of interface mappings, as we need to specify where the adaptor is to get the value of any parameter that gets synthesized. An interface mapping is *unambiguous* iff each parameter appears in at most one right-hand side of a `<parm_mapping_rule>` rule. We assume that all interface mappings are complete and unambiguous.[6]

4.2 Validity of Adaptors with Respect to Interface Mappings

Let $\mathcal{I}$ be an interface mapping between two collaboration specifications C_1 and C_2 supporting protocols P_1 and P_2 respectively. An adaptor $\mathcal{A}$ is *valid* w.r.t. $\mathcal{I}$ iff

(1) $\mathcal{A}$ is well formed w.r.t. P_1 and P_2 and

(2) $\mathcal{A}$ is correct w.r.t. $\mathcal{I}$ (i.e., $\mathcal{A}$ satisfies the constraints given in $\mathcal{I}$).

LEMMA 4.2.1. *There exists an algorithm for checking whether an adaptor is valid w.r.t. an interface mapping.*

[6]Ambiguous interface mappings can make sense; for instance, one may want to specify two ways to synthesize a particular parameter of a message, depending upon what previous messages have been received. For simplicity, we do not discuss ambiguous interface mappings in this article.

```
//Parameter mapping rules
bidder::newBid.bidId    -> auctioneer::bid.bidderId;
bidder::newBid.myBid    -> auctioneer::bid.amount;
bidder::requestToBid.bidItem  -> auctioneer::bid.itemBiddingOn;
auctioneer::newItem.itemDescr -> bidder::auctionBegin.auctionInfo;
auctioneer::newItem.bidderId  -> bidder::canBid.bidId;
auctioneer::update.highBid    -> bidder::newHighBid.price;
// Parameter usage rules
one-shot(bidder::newBid.bidId);
one-shot(bidder::newBid.myBid);
one-shot(auctioneer::newItem.itemDescr);
one-shot(auctioneer::newItem.bidderId);
one-shot(auctioneer::update.highBid);
// Causality rules
forward auctioneer::newItem as bidder::auctionBegin;
forward auctioneer::update as bidder::newHighBid if possible;
forward auctioneer::rejectBid as bidder::bidNotOk;
forward auctioneer::acceptBid as bidder::bidOk;
forward auctioneer::itemSold as bidder::gavel;
if (bidder::reqToBid) then forward auctioneer::update as bidder::newHighBid;
forward auctioneer::update as auctioneer::updateAck;
forward bidder::requestToBid as bidder::canBid;
forward bidder::newBid as auctioneer::bid;
(!bidder::cannotBid)*;
```

Fig. 6. An interface mapping for the Auctioneer and Bidder protocols.

The proof of this lemma follows from Theorem 3.5.2 and Lemmas 3.5.3 and 3.5.4.

Example 4.2.1. Figure 6 gives an interface mapping for the Auctioneer and Bidder collaboration specifications. The last rule states that the adaptor should never send the `cannotBid` message.

4.3 Synthesis of Adaptors

In this section we describe an algorithm that, given two protocols P_1 and P_2 for two collaborating components C_1 and C_2 and an interface mapping $\mathcal{I}$ mapping between the protocols, will either construct an adaptor that is valid w.r.t. $\mathcal{I}$ or will conclude that no such adaptor exists.

Assume that there are n causality constraints in $\mathcal{I}$ and that $\mathcal{I}$ dictates that the adaptor contains the memory cells $Cells$.[7] Let $PatternState_i$ $(1 \leq i \leq n)$ be the states of the pattern automaton for the pattern given in the ith causality constraint. Let $PossibleAdaptorStates$ be the set of all tuples of the form $States(P_1) \times States(P_2) \times 2^{Cells} \times PatternState_1 \times \cdots \times PatternState_n$. Each state of the constructed adaptor will be an element of $PossibleAdaptorStates$ or the special state *error*. We will maintain the invariant that an adaptor enters state $\langle s_1, s_2, memCells, p_1, \cdots, p_n \rangle$ during a collaboration with components C_1 and C_2 iff at this point in time component C_1 is in protocol state s_1; component C_2 is in protocol state s_2; the adaptor memory cells in $memCells$ are valid; and the pattern

[7]Recall that an adaptor has one memory cell for each parameter it can receive. Hence an examination of the interface mapping (or the component protocols) will indicate the set of memory cells in the adaptor.

automaton for the ith causality constraint is in state p_i. The adaptor will enter the state *error* iff the collaboration has violated a causality constraint.

Let $u = \langle s_1, s_2, memCells, p_1, \cdots, p_n \rangle$ and $u' = \langle s'_1, s'_2, memCells', p'_1, \cdots, p'_n \rangle$ be two $PossibleAdaptorStates$. We say that the adaptor transition $u : m \rightarrow u'$ is *enabled* iff all of the following are satisfied:

(1) $(s_1 : m \rightarrow s'_1) \in Transitions(P_1)$ and $s_2 = s'_2$ *or* $s_2 : m \rightarrow s'_2 \in Transitions(P_2)$ and $s_1 = s'_1$;
(2) if m is a message to be received by the component (and therefore to be sent by its mate the adaptor) then $memCells$ contains all the cells required to synthesize the parameters of m, as specified by $\mathcal{I}$;
(3) if m is a message to be received by the component (and therefore to be sent by its mate the adaptor) then $memCells'$ is the same as $memCells$ except that if $\mathcal{I}$ specifies that $parm$ has the one-shot property in $\mathcal{I}$, and if $Cells.parm$ is used to synthesize a parameter of m, then $Cells.parm$ is not in $memCells'$;
(4) if m is a message to be sent by the component (and therefore to be received by its mate the adaptor) then $memCells'$ is the same as $memCells$ except that for each parameter $parm$ of m, $memCells'$ additionally contains $Cells.parm$;
(5) the pattern automaton of the ith ($1 \leq i \leq n$) causality constraint moves from state p_i to state p'_i upon the reception/sending of message m. Furthermore, p'_i *cannot* equal *error*.

The first requirement guarantees that an adaptor transition is only enabled if the message can arrive from or can be sent to one of the collaborating components without causing an unspecified reception. The second requirement guarantees that the transition will only be enabled if the adaptor has enough information to synthesize the parameters of the message it is sending. The third and fourth requirements guarantee that the transition will take the adaptor to an adaptor state which correctly specifies the valid memory cells of the adaptor after this transition is taken. The last requirement guarantees that the transition will take the adaptor to an adaptor state which correctly specifies the pattern automaton states of the adaptor after this transition is taken. A resulting pattern automaton state cannot be *error*, as that would violate its causality constraint.

In specifying an adaptor transition, besides specifying the receipt or sending of a message, one also needs to specify how parameters are synthesized and which memory cells are written or invalidated. Since the interface mapping specifies how parameters are to be synthesized, and the adaptor constructed by the synthesis algorithm will always be valid w.r.t. this mapping, and since the adaptor state specifies which memory cells are valid, this information is implicit in the transition, and we will not explicitly state this additional information.

The synthesis algorithm will construct the adaptor in phases. In phase 1 it will construct an initial adaptor A_1. In each successive phase it will remove some of the states and transitions from the adaptor produced by the preceding phase, thereby forming a new adaptor. The algorithm will finally reach a fixed point where no more states and transitions can be removed. If the resulting adaptor is empty — it is the null adaptor — then no valid adaptor exists. Otherwise the resulting adaptor is valid w.r.t. $\mathcal{I}$. It will satisfy each causality constraint using weak satisfaction semantics.

Phase 1 proceeds as follows. A worklist is initialized to contain a single state, the initial adaptor state $\langle init_{P_1}, init_{P_2}, \emptyset, p_{1_{init}}, \cdots, p_{n_{init}} \rangle$, where $p_{i_{init}}$ is the initial state of the pattern automaton of the ith causality rule. This initial state is marked. The algorithm then continuously performs the following actions, until the worklist is empty.

Remove a state u from the worklist. For each enabled transition from u to u', where $u' \in PossibleAdaptorStates$, add this transition to the adaptor. If u' is not already marked, add u' to the worklist and mark it.

When no more states are on the worklist, then this first phase of the algorithm concludes. The adaptor A_1 will consist of the marked states and the transitions added during the phase.

As shown in the Appendix, A_1 captures all of the "good" collaborations that can possibly occur between the collaborating components and any adaptor that is valid w.r.t. $\mathcal{I}$. The problem is that A_1 may also contain "bad" collaborations — that is, collaborations that deadlock or that have unspecified receptions. In order to get rid of these collaborations, we need to remove states and transitions that allow these bad collaborations to occur.

This is accomplished by the following phases of the algorithm. Given adaptor A_j, $j \geq 1$, phase $j+1$ first checks to see if A_j contains a deadlock state or an unspecified reception state u. In the former case, u is a nonfinal state with no outgoing transitions. In the latter case, u specifies that one of its collaborating components is in a send state s (*not* a mixed state) and can send message m, but there is no adaptor transition from u that accepts m. If such a state u exists, then that state is removed from the adaptor with all of its transitions. The resulting adaptor, A_{j+1}, is the adaptor produced by phase $j+1$.

The algorithm terminates when either the adaptor constructed by a phase has no more deadlock or unspecified reception states, or the adaptor is empty — it contains no states. In the latter case we can conclude that there is *no* adaptor that is valid w.r.t. $\mathcal{I}$. Otherwise this adaptor is a valid adaptor for this interface mapping. The proof of the following theorem is given in the Appendix.

THEOREM 4.3.1. *The adaptor synthesis algorithm will either produce a valid adaptor w.r.t. the interface mapping $\mathcal{I}$ or will correctly conclude that no such adaptor exists.*

Given an adaptor specification synthesized by this algorithm, it is almost completely straightforward to automatically synthesize the code that implements the adaptor. Of course, the adaptor synthesized by the algorithm will use abstract protocols. As discussed toward the end of Section 3.5, these protocols need to be augmented with synchronization constraints to be able to work with an arbitrator. The only complication is that the specification is not completely deterministic; the adaptor specification produced can contain mixed states, where the adaptor is allowed to send or receive a message, and even send states allow the adaptor to arbitrarily choose one of possibly many messages to send. In these cases, the code implementing the adaptor can just make an arbitrary decision of which path to follow. Another approach is to produce an adaptor that does not have this nondeterminism. To this end we say that an adaptor specification is *strongly deterministic* iff (1) it has no mixed states and (2) every send state has only one outgoing

transition. It is not hard to show that if there exists a valid adaptor, then there exists a strongly deterministic valid adaptor.

5. RELATED WORK

The contributions of this article fall into four categories: augmenting interface descriptions with protocols, protocol compatibility, the semiautomatic generation of software adaptors, and component composition.

Adding protocols to interface specifications is based upon the theory that additional constraints on interface descriptions can help in the task of composing software. It helps the programmers of modules by allowing tools to automatically check that the module obeys the constraints given in the specification, and it helps users of the module by formally documenting usage assumptions. Previous work by the authors on typestate [Russell et al. 1994; Strom and Yellin 1993; Strom and Yemini 1986] shows how to statically check modules for compliance to such constraints. Other recent work on adding constraints to interface descriptions includes the Rapide system [Luckham et al. 1995].

There have been other proposals for augmenting interface descriptions with sequencing constraints. One of the earliest was Campbell and Habermann's *path expressions* [Campbell and Habermann 1974]. A path expression declares sequencing constraints on method invocations and can be used to limit the number of concurrent method executions. The emphasis of this work is concurrency control and synchronization within a server; the path expression does not limit the order in which clients can invoke methods, but determines how the scheduler orders these invocations. Furthermore, the path expression does not define a relationship between a particular client and server, but between any number of clients and servers.

Procol [van den Bos and Laffra 1991] incorporates *protocols* into its object descriptions. These are closer to our own notion of protocol, as they allow a server to specify interactions with a particular client. However, a protocol in Procol is less abstract than our own protocols, as it relates messages to internal implementations, whereas our protocols describe only externally visible interface properties. Furthermore, like path expressions, Procol only expresses a server's protocol (i.e., receives), not a client's protocol (i.e., sends).

Whereas the preceding systems use protocols as a run-time device to control client/server interactions, we propose protocols to augment the type system and control legal bindings (or at least to warn of potential conflicts). To this end we defined protocol compatibility. A similar approach is taken by Nierstrasz [1993] in his paper on regular types for active objects. However, this work once again focuses on a client/server model where servers only specify receives while clients only specify sends. There are other technical differences between our approaches, including how to handle nondeterminism.

Our notion of protocol compatibility is different from that used in the communicating finite-state machine literature [Brand and Zafiropulo 1983]. Using that model, for instance, a protocol that first sends message a and then receives b is compatible with a mate that first sends message b and then receives a. According to our definition, the protocols are not compatible. On the other hand, in our model a protocol P is always compatible with its inverse $\overline{P}$, which is not true for the asynchronous communicating finite-state machine model. Without any bounds on the

lengths of queues, and without any synchronization guarantees, it is in general *undecidable* to determine whether two communicating finite-state machines will contain no protocol errors such as deadlock [Brand and Zafiropulo 1983]. Many researchers have investigated what restrictions need to be placed on the protocols to guarantee the decidability of the problem [Brand and Zafiropulo 1983; Gouda et al. 1984; 1987].

We have introduced arbitrators and our new definition of protocol compatibility for several reasons. First, we did not want to deal with an undecidable problem or even a computationally hard one. Second, we wanted a model that made it easy for people and tools to use and reason about protocols. We considered it important to be able to use a protocol $\overline{P}$ to collaborate with a component using a protocol P.

Our approach is similar to, but more general than, that given by Gouda et al. [1984]. We are more general in that we allow for mixed states (which requires our introduction of arbitrators), and we allow for one protocol to receive messages that cannot be sent by the other protocol as long this will not result in deadlock. We also allow one protocol to send messages that cannot be received by the other as long as the states from which these messages are sent are unreachable during any collaboration involving these two protocols.

The other area addressed by our article is that of software adaptors. This has been studied by several other researchers using various different frameworks. Wiederhold [1992] describes a general framework for software *mediators*. In Purtillo et al.'s *Polylith* system, a language called *NIMBLE* has been developed that allows programmers to easily state mappings between a client invocation of a function and that invocation on the server [Purtilo and Atlee 1991]. Supported transformations include reorderings of parameters, data type conversions on parameters, and specification of default values for missing parameters. This work is important in making client/server applications work when one can express a 1-1 mapping between client calls and server invocations. This work is inadequate, however, when dealing with stateful adaptors where (1) multiple calls need to be mapped into a single call, (2) synchronization messages need to be synthesized, etc. In the words of Konstantas [1993], "By reducing the interoperability 'interface' to the level of a procedure call, the inter-relation of the interface procedures is lost, since the interoperability support no longer sees the interface as a single entity but as isolated procedures."

The object-oriented interoperability work of Konstantas [1993], the work on gluons by Pintado et al. [1992], and the interface adaptors of Thatte [1994] are similar to ours in that they focus on translation of interfaces (objects). Yet none of these approaches use protocols as the glue to hold together the messages of an interface. In this regard, our work is most similar to protocol conversion [Lam 1988; Okumura 1986; Shu and Liu 1989]. In these works, like our own, protocols are used to specify interfaces; some sort of specification is given to define the relationship between different protocols; and an algorithm is described that synthesizes a converter given the protocols and the specification. This article extends these works in several ways. Most significantly, these works do not allow messages to have parameters; their focus is on synchronizing messages of one protocol with messages in the other protocol. Because their model lacks parameters, their adaptors consist of a state (corresponding to the states of its two mates) and a message queue, unlike our formulation where an adaptor has additional memory used to store parameters.

Although one may be able to mimic parameters as separate messages, our formulation is much more natural (at least to objected-oriented systems). It also allows us to model many more properties of adaptors, such as parameter translation, reuse of parameters in multiple messages, and memory consistency of adaptors.

The role of parameters in our model goes beyond expressibility, as it also plays a fundamental role in our synthesis algorithm. This is because when the model does not contain parameters, then there are no natural constraints on the adaptor — one can always construct an adaptor that synthesizes messages to send to either component whenever that component's protocol requires such a message to be received. The role of the specification is to put constraints on adaptor behavior so that not all compatible adaptors are valid. When the model includes parameters and parameter-mapping rules, however, then there exist natural constraints on the adaptor; the adaptor can only synthesize a message if it has already received sufficient information (in the form of received parameters) to synthesize the parameters of this message. In this regard, Shu and Lius' [1989] mapping set is somewhat similar to our parameter-mapping specifications, except that they map messages of one protocol to messages in the other protocol, whereas we map parameters to parameters. This provides a finer level of granularity in the mapping that is important in many applications.

Whereas parameter-mapping rules provide a natural way to express the semantics of the adaptor, we recognize that additional constraints are still often needed to specify additional properties that the adaptor must satisfy. This is the reason that our interface mappings also allow one to express causality constraints. These constraints allow one to specify properties similar to the sorts of properties expressible by Okimura's [1986] conversion seed.[8]

Overall, in both goals and methodology of building a robust model for component composition, the work by Allen and Garlan [1994a; 1994b] on specification of ports and connectors is the most closely related to ours. In both models, components may have one or more interfaces, each with its own formal specification based on finite-state protocols — Allen and Garlan's interfaces are called *ports*. Whereas we distinguish between the cases where two interface specifications can be directly bound and those where an adaptor is required, in Allen and Garlan's model there is always an adaptor-like *connector*. Their connectors are first-class, reusable components in their own right and can support n-party interactions. Connectors are polymorphic in that any component's port whose protocol is compatible with a given connector's role can be plugged into that role. Allen and Garlan are able to use model-checking tools to verify both compatibility and deadlock freedom. Their port and role protocols are defined in a subset of CSP, including the nondeterministic and deterministic choice operators ($\Box$ and $\sqcap$). Our model imposes additional restrictions on the interfaces that may be specified: in particular, a component's interface may only make a deterministic choice involving zero or

[8]It is interesting to note that causality rules are needed to specify liveness properties of adaptors. The parameter-mapping rules only indicate under what conditions an adaptor is *permitted* to send a message; it does not indicate when the adaptor *must* send a message. (Sometimes the combination of a parameter-mapping rule and the fact that the adaptor cannot deadlock will have the effect of enforcing a liveness property, but this is not always the case.)

more input events and an optional nondeterministic choice among output events. Deterministic choices among output events (where the environment dictates which message is output) and nondeterministic choices among input events (where the recipient determines which message he will accept) are not allowed. Although more restricted, our interface semantics is still adequate to capture the communications models of most object-based systems that use either method calls or asynchronous message passing. Our safety criterion is also stronger: we require that the two parties agree on the order of sent and received messages. Although adaptors can be used to bypass this strict agreement on the order of messages, it is still more restrictive than the sorts of protocols that are compatible under the usual asynchronous semantics, as discussed in Section 3.4. These restrictions permit us to simplify the analysis of compatibility, to automatically produce arbitrators to manage contention, and to synthesize adaptors when the protocols are not syntatically compatible but are functionally compatible.

One way in which our work can be generalized is to support not only two-party interactions, but multiple-party interactions, as in Allen and Garlan's work cited above, as well as in the work on contracts by Helm et al. [1990]. In this case, the "adaptor" is not just an entity that bridges the differences between interfaces, but a more general mechanism for "gluing" together multiple applications into a composite application. It would serve the role as coordinator and would probably require a more general scripting mechanism than the adaptors of this article.

The component composition model used in this article allows an interface in one component to be bound to an interface in a second component. It does not allow an interface in one component to be bound to multiple interfaces (in several components). Additional research is needed to explore the protocol semantics of such compositions.

Another area for future research is to investigate richer languages for expressing protocols. Such languages should include notions of fairness and liveness.

APPENDIX

PROOFS OMITTED FROM MAIN TEXT

Dataflow Analysis of Collaborations

In order to prove Lemmas 3.5.1, 3.5.3, and 3.5.4, we first present a generic tool for checking whether a property will hold for any collaboration involving a particular adaptor. To help in this task, we utilize techniques similar to classical dataflow algorithms developed for program analysis and optimization[Kam and Ullman 1977; Kildall 1973].

Given protocols P_1 and P_2, and adaptor $\mathcal{A}$, we build a *collaboration graph* representing all possible collaborations. There will exist one node in the graph for each tuple $\langle s, t, u\rangle$ (of type $States(P_1) \times States(P_2) \times States(\mathcal{A})$) such that there exists some collaboration history with a collaboration state of the form $\langle s, t, u\rangle$. As shown in the proof of Theorem 3.3.1, the set of all such tuples is finite and computable. There will exist an edge from a node representing $\langle s, t, u\rangle$ to a node representing $\langle s', t', u'\rangle$ iff, in some collaboration history, a collaboration state $\langle s, t, u\rangle$ can immediately precede a collaboration state $\langle s', t', u'\rangle$. The label of an edge will contain a set of message names. In particular, the label will contain the name of any mes-

sage that can be sent to or received from the adaptor if that message causes the transition from the predecessor to the successor state. The resulting graph can be cyclic. Any path in the collaboration graph starting at $\langle init_{P_1}, init_{P_2}, init_A \rangle$, the entry node of the graph, represents a collaboration history $\alpha \in Collabs(P_1, P_2, \mathcal{A})$.

Let *Props* be some finite set of properties about the collaboration. The goal of the analysis is to label each node of the graph with a subset P of *Props* such that $\langle s, t, u \rangle$ is labeled by P iff the following invariant is maintained: $p \in P$ iff there exists $\alpha \in Collabs(P_1, P_2, \mathcal{A})$ such that $\alpha = \cdots \alpha_i \cdots$, $\alpha_i = \langle s, t, u \rangle$, and p is true of the collaboration when it enters state α_i. That is, the label at a node $\langle s, t, u \rangle$ will include an element p of *Props* iff there exists some collaboration state $\langle s, t, u \rangle$ of some collaboration such that p is true when the collaboration enters that state.

Formally, each label is drawn from an element of the semilattice L. L consists of a set of properties (i.e., an element of 2^{Props}) and a *meet* operation $\cup$ (set union). The least element is *Props*; the greatest element is $\emptyset$; and the partial order is $\supseteq$ (set inclusion), i.e., $a \geq b \Rightarrow a \subseteq b$. Each edge of the graph has an associated function $f : 2^{Props} \rightarrow 2^{Props}$ that shows how to derive a set of properties at a successor state from the set of properties at the predecessor state. For monotonic frameworks, this function must be monotonic; $S_1 \subseteq S_2 \Rightarrow f(S_1) \subseteq f(S_2)$. In our case, for each function f there will exist a function $f' : Props \rightarrow Props$ such that $f(S) = \cup_{s \in S} f'(s)$. Hence f will satisfy the property that $f(S_1 \cup S_2) = f(S_1) \cup f(S_2)$. In technical terms, this means that this framework is not only monotonic, it is *distributive* [Kam and Ullman 1977].

Property computation of the collaboration graphs proceeds as follows. Initially each node is labeled by $\emptyset$. Repeatedly a node of the graph is evaluated until a fixed point is reached. Evaluation of a node means computing the value at this node by applying the edge function to each predecessor node label and taking the meet of these values to form the label at this node. Because this framework is monotonic, we are guaranteed to reach a fixed point. Furthermore, because this framework is distributive, we are guaranteed to compute the *meet-over-all-paths* solution [Kam and Ullman 1977]. This means that for each node representing $\langle s, t, u \rangle$ computed to have label P, there exists $p \in P$ iff there exists $\alpha \in Collabs(P_1, P_2, \mathcal{A})$ such that $\alpha = \cdots \alpha_i \cdots$, $\alpha_i = \langle s, t, u \rangle$, and p is true of the collaboration when it is in state α_i.

With this introduction, we can now prove Lemmas 3.5.1, 3.5.3, and 3.5.4. In each case we will apply the framework given above by implicitly defining the functions f and f'.

Proof of Lemma 3.5.1. To check for memory consistency we use the dataflow analysis algorithm given above. In this case, $Props = 2^{Cells}$. If $\langle s, t, u \rangle$ is labeled by $P \subseteq 2^{Cells}$, then for each set of memory cells $memCells \in P$, there exists some collaboration history passing through the adaptor state u such that the valid memory cells at u exactly equal *memCells*.

In order to use the dataflow analysis algorithm given above, we need to define how to compute the label P' at a successor state $\langle s', t', u' \rangle$ given the label P of the predecessor state $\langle s, t, u \rangle$. This is straightforward: for each set $memCells \in P$, and for each message m that labels the edge from $\langle s, t, u \rangle$ to $\langle s', t', u' \rangle$, if the corresponding adaptor transition is a receive transition with save-actions, then the

newly received parameters are added to the set of valid memory cells $memCells$. If the corresponding adaptor transition is a receive or send transition with invalidate-actions, then the invalidated parameters are removed from the set of valid memory cells $memCells$. The resulting set is added to P'.

Once we have computed the labels for each node of the graph, for each node corresponding to state $\langle s, t, u\rangle$ with label P, we check each transition from this state to a successor state in which the adaptor sends a message m. It must be the case that for each $memCells \in P$, $memCells$ contains all the cells needed to synthesize the parameters of m. If this fails to be the case, the adaptor is not memory consistent; otherwise it is. □

Proof of Lemma 3.5.3. We can use a slight variation of the proof just given to determine whether a particular parameter has the `one-shot` property. In this case, we run the algorithm given above, but we also add an action to invalidate $Cells.p$ in any adaptor transition that uses $Cells.p$ to synthesize a parameter. If the resulting adaptor is still memory consistent, then the adaptor satisfies `one-shot(p)`, and otherwise it does not. □

Proof of Lemma 3.5.4. We need to show how to check whether an adaptor satisfies a pattern. We begin with weak satisfaction. For any message pattern we can construct a finite automaton that recognizes that pattern. During a collaboration, every transition will cause this automaton to advance to a new state, based upon which message was sent from or received by the adaptor in this transition. We will assume that every such pattern automaton also has a special state called *error* that will be entered during a collaboration if a transition occurs that violates the pattern. (For instance, if the automaton corresponds to the pattern `m1 . m2` and after receiving an `m1` message an `m3` message is sent, the pattern automaton would enter the state *error*.) Once it has entered the *error* state, the pattern automaton never leaves this state.

In order to determine whether the adaptor weakly satisfies a pattern, we use the dataflow algorithm given above. In this case, $Props$ will be the states of the pattern automaton. The meaning of labeling a node by a subset of automaton states is as follows: node $\langle s, t, u\rangle$ includes the automaton state p as part of its label iff there exists some collaboration state $\langle s, t, u\rangle$ of some collaboration history such that when that collaboration state is entered, the automaton recognizing the given pattern will be in the automaton state p.

Computing the properties at a successor node $\langle s', t', u'\rangle$ from a predecessor node $\langle s, t, u\rangle$ labeled P is as follows: For each $p \in P$, for each message m that labels the edge from $\langle s, t, u\rangle$ to $\langle s', t', u'\rangle$, the pattern automaton is consulted to see what state p' the pattern automaton enters when starting in state p and making the transition involving message m. This state p' is then added to the label of the successor state $\langle s', t', u'\rangle$. We apply the dataflow analysis algorithm to label each node of the collaboration graph.

Once we have computed the labels at each collaboration graph node, we check that no node has a label that contains the *error* symbol and that any final state of the collaboration graph is labeled only by final states of the pattern automaton. If this is the case then the adaptor weakly satisfies the pattern; otherwise it does not.

Checking for strong satisfiability requires more work. In addition to the above

requirements, strong satisfiability means that whenever the adaptor enters a collaboration state that does not recognize the pattern, it will eventually enter a collaboration state that will recognize the pattern (i.e., a final state of the pattern automaton). This means that we need to check all cycles of the collaboration graph and make sure that any collaboration that forever traverses the cycle will infinitely often enter a state that recognizes the pattern. If there exists some node of the cycle that is labeled only by terminal states of the pattern automaton, then this is certainly true. Otherwise, we need to check if the condition is satisfied by using the following algorithm:

Consider each node n on the cycle, and each nonfinal pattern automaton state p labeling that node. The fact that p labels the node means that there exists some collaboration that enters this node with the pattern automaton in state p. Say that the pattern automaton has k states. Consider all sequences of transitions that cause the collaboration to traverse the cycle k times, starting at node n and returning to node n. Let S be the set of these sequences. There exists a finite (although possibly exponential) number of such sequences. (There exist more than one such sequence, since each edge on the cycle can be labeled by multiple message names.) Consider each sequence in S. Starting at node n, simulate the traversal given by this sequence. Initially, let n be labeled by p, and all other nodes on the cycle have an empty label. Each time a node is entered, compute the new pattern automaton state, and add that state to the label at that node.

When adding a pattern automaton state to the label at a node, consider the following three possibilities: (1) If the newly computed pattern automaton state is a final state, then stop. It means that using this sequence, the collaboration eventually entered a final state, and using this sequence the adaptor strongly satisfies the pattern. Continue by checking the next sequence in S. (2) If the newly computed automaton state p' at node n' is the same as an existing label p' at that node, then stop. It means that along this collaboration sequence, the collaboration can begin at node n' with the automaton in state p' and eventually return to node n' with the automaton once again in state p' without the automaton entering a final state in the meantime. Hence we can construct an infinite collaboration with this behavior (never entering a final automaton state) by repeating this sequence prefix an infinite number of times. Therefore the adaptor does not strongly satisfy the pattern. (3) The label is a nonfinal pattern automaton state that does not already label this node. Continue with simulating this sequence.

Since each sequence traverses the cycle k times, since node n is originally labeled by an automaton state, and since there are only k automaton states, one of the first two conditions will be true by the time we finish simulating the sequence and labeling node n with the $k + 1$st pattern automaton state. If for any sequence we detect that the adaptor will not strongly satisfy the pattern, then we can conclude that the adaptor does not strongly satisfy the pattern. Otherwise we can conclude that the adaptor does strongly satisfy the pattern w.r.t. this cycle. If this is true for all cycles, we can conclude that the adaptor strongly satisfies the pattern. □

Proof of Theorem 4.3.1. If the synthesis algorithm of Section 4.3 produces a nonnull adaptor A, then the resulting adaptor will certainly be well formed w.r.t. P_1 and P_2, as all unspecified reception and deadlock states have been removed. To

show memory consistency, as well as correctness w.r.t. the interface mapping $\mathcal{I}$, we need to show that, for every $\alpha \in Collabs(P_1, P_2, A)$, α is memory consistent and that it satisfies each <parm_mapping_rule>, each <Parm_usage_rule>, and <causality_rule>. This can be shown by induction on the length of α.

To prove the theorem we need to also show that if a null adaptor is produced by the algorithm, then no valid adaptor exists.

For any collaboration history α between an adaptor and two components, let $Messages(\alpha)$ be the sequence of messages that are exchanged by the components and the adaptor as specified in the collaboration history α. $Messages(\alpha_1) = Messages(\alpha_2)$ iff $\alpha_1 = \alpha_2$.

Recall that A_i is the adaptor produced by phase i of the algorithm. Assume that $\mathcal{I}$ admits a valid adaptor, and let B be any valid adaptor w.r.t. to $\mathcal{I}$. We show that the following invariant is maintained: if $\beta \in Collabs(P_1, P_2, B)$ then there exists $\alpha \in Collabs(P_1, P_2, A_i)$ where $Messages(\alpha) = Messages(\beta)$. We prove this claim by induction on A_i, where i is the phase of the algorithm. For the base case, we can show that this is true for A_1 using induction on the length of β.

Assume that the invariant is true for adaptor A_i. The construction of A_{i+1} removes some state u from the adaptor where u is either a deadlock or an unspecified reception state. Obviously, $Collabs(P_1, P_2, A_{i+1}) \subset Collabs(P_1, P_2, A_i)$. If our claim is wrong, then there exists a collaboration $\beta \in Collabs(P_1, P_2, B)$ and $\alpha \in Collabs(P_1, P_2, A_i)$ such that $Messages(\alpha) = Messages(\beta)$, but where $\alpha \notin Collabs(P_1, P_2, A_{i+1})$. It must be that α passes through the removed state $u = \langle s_1, s_2, memCells, p_1, \cdots, p_n \rangle$. Consider the following two cases: (1) u is a deadlock state in A_i or (2) u is an unspecified reception state in A_i. In each case we will show that the collaboration histories β and α imply that there exist some other collaboration histories β' and α' in $Collabs(P_1, P_2, B)$, and $Collabs(P_1, P_2, A_i)$ respectively, and that $\alpha' \in Collabs(P_1, P_2, A_i)$ contradicts the assumption that u is a deadlock or unspecified reception state in A_i.

Regarding case (1), since u is a deadlock state, it must be that $\alpha = \alpha_1 \rightarrow_{m_1} \cdots \rightarrow_{m_{n-1}} \alpha_n$ where $\alpha_n = \langle s_1, s_2, u \rangle$, and that s_1 and s_2 are not both final states. Since $Messages(\beta) = Messages(\alpha)$, it must be that $\beta = \beta_1 \rightarrow_{m_1} \cdots \rightarrow_{m_{n-1}} \beta_n$, and that when adaptor B is in state β_n, its collaborating components are in states s_1 and s_2, respectively. Since s_1 and s_2 are not both final states, and B is compatible with P_1 and P_2, it must be that there exists some collaboration $\beta' = \beta_1 \rightarrow_{m_1} \cdots \rightarrow_{m_{n-1}} \beta_n \rightarrow_{m_n} \beta_{n+1} \cdots \in Collabs(P_1, P_2, B)$. By induction there exists $\alpha' \in Collabs(P_1, P_2, A_i)$ such that $Messages(\alpha') = Messages(\beta')$. Obviously, $\alpha' = \alpha_1 \rightarrow_{m_1} \cdots \rightarrow_{m_{n-1}} \alpha_n \rightarrow_{m_n} \alpha_{n+1} \cdots$. But then u is not a deadlock state in A_i, which contradicts our assumption.

Regarding case (2), let $\alpha = \alpha_1 \rightarrow_1 \cdots \rightarrow_{m_{n-1}} \alpha_n \cdots$ where $\alpha_n = \langle s_1, s_2, u \rangle$, and let $\beta \in Collabs(P_1, P_2, B)$ such that $Messages(\alpha) = Messages(\beta)$. Since $u = \langle s_1, s_2, memCells, p_1, \cdots, p_n \rangle$ is an unspecified reception state, it must be that there exists some message m' such that when one of the components, say C_1, is in the send state s_1, it can send the message m', but the adaptor A_i has no transition from u to another state that allows it to receive m'. Let $MessageHistory = Messages(\alpha_1 \rightarrow_{m_1} \cdots \rightarrow_{m_{n-1}} \alpha_n) \cdot m'$. $MessageHistory$ is the sequence of messages in α up to the point where α enters state u and is then followed by the message m'. Since B is a valid adaptor, and since by assumption $\beta \in Collabs(P_1, P_2, B)$,

it must be that there exists a collaboration history $\beta' \in Collabs(P_1, P_2, B)$ such that $Messages(\beta') = MessageHistory$. By induction there exists $\alpha' \in Collabs(P_1, P_2, A_i)$ such that $Messages(\alpha') = Messages(\beta')$. Obviously, $\alpha' = \alpha_1 \rightarrow_{m_1} \cdots \rightarrow_{m_{n-1}} \alpha_n \rightarrow_{m'} \alpha_{n+1}$. But then u does have a transition upon receipt of an m' message in A_i, which contradicts our assumption.

Hence the adaptor A_k produced by the final phase of the algorithm will have the property that if $\beta \in Collabs(P_1, P_2, B)$, then there exists $\alpha \in Collabs(P_1, P_2, A_k)$ where $Messages(\alpha) = Messages(\beta)$. In particular, if A_k is the null adaptor, it must be that no valid adaptor B exists. □

ACKNOWLEDGEMENTS

We thank Josh Auerbach, who first pointed us toward the protocol conversion literature and put much of its work into context for us. We also wish to thank David Garlan, who provided many valuable comments in improving the presentation of an earlier version of this article. Finally, the anonymous referees provided many useful suggestions.

REFERENCES

Allen, R. and Garlan, D. 1994. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*. IEEE, New York.

Arjomandi, E., O'Farrell, W., Kalas, I., Koblents, G., Eigler, F., and Gao, G. 1995. ABC++: Concurrency by inheritance in C++. *IBM Syst. J. 34,* 1.

Arnold, K. and Gosling, J. 1996. *The Java Programming Language*. Addison Wesley, Reading, Mass.

Auerbach, J. S., Goldberg, A. P., Goldszmidt, G. S., Gopal, A. S., Kennedy, M. T., Rao, J. R., and Russell, J. R. 1994. Concert/C: A language for distributed programming. In *Winter 1994 USENIX Conference*. USENIX Assoc., Berkeley, Calif.

Brand, D. and Zafiropulo, P. 1983. On communicating finite-state machines. *J. ACM 30,* 2 (Apr.), 323–342.

Brockschmidt, K. 1994. *Inside OLE2*. Microsoft Press, Redmond, Wash.

Campbell, R. H. and Habermann, A. N. 1974. *The Specification of Process Synchronization by Path Expressions*. Vol. 16. Springer-Verlag, Berlin, 89–102.

Digitalk. 1993. *PARTS Workbench User's Guide*. Digitalk, Sunnyvale, Calif.

Garlan, D., Allen, R., and Ockerbloom, J. 1995. Architectural mismatch or why its hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*. IEEE, New York.

Gouda, M. G., Gurari, E. M., Lai, T., and Rosier, L. E. 1987. On deadlock detection in systems of communicating finite state machines. *Comput. Artif. Intell. 6,* 3, 209–228.

Gouda, M. G., Manning, E. G., and Yu, Y. T. 1984. On the progess of communication between two finite-state machines. *Inf. Control 63*, 200–216.

Huynh, T., Jutla, C., Lowry, A., Strom, R., and Yellin, D. 1994. The global desktop: A graphical composition environment for local and distributed applications. In *Programming Technology Forum*, R. Pinter, Ed. IBM, Armonk, N.Y.

IBM. 1993. *SOMobjects Developer Toolkit Users Guide, Version 2.0*. IBM, Armonk, N.Y.

IBM. 1994. *VisualAge User's Guide and Reference*. IBM, Armonk, N.Y.

Kam, J. B. and Ullman, J. D. 1977. Monotone data flow analysis frameworks. *Acta Inf. 7*, 305–317.

Kildall, G. A. 1973. A unified approach to global program optimization. In *The 1st ACM Symposium on Principles of Programming Languages*. ACM, New York, 194–206.

Konstantas, D. 1993. Object oriented interoperability. In *Visual Objects*, D. Tsichritzis, Ed. Universite De Geneve, Switzerland. Also Appeared in ECOOP 93.

Lam, S. S. 1988. Protocol conversion. *IEEE Trans. Softw. Eng. 14,* 3 (Mar.), 353–362.

Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D., and Mann, W. 1995. Specification and analysis of system architecture using Rapide. *IEEE Trans. Softw. Eng. SE21,* 4 (Apr.), 336–355.

Nierstrasz, O. 1993. Regular types for active objects. In *OOPSLA '93 Conference Proceedings. ACM SIGPLAN Not. 25*, 10 (Oct.).

Okumura, K. 1986. A formal protocol conversion method. In *Proceedings of the ACM SIGCOMM '86 Symposium.* ACM, New York, 30–37.

Pintado, X. and Junod, B. 1992. Gluons: Support for software component cooperation. In *Object Frameworks*, D. Tsichritzis, Ed. Universite De Geneve, Switzerland.

Purtilo, J. M. and Atlee, J. A. 1991. Module reuse by interface adaption. *Softw. Pract. Exper. 21,* 6 (June).

Russell, J., Strom, R. E., and Yellin, D. M. 1994. A checkable interface language for pointer-based structures. In *ACM Workshop on Interface Languages. SIGPLAN Not.* 29, 8 (Aug.).

Shu, J. and Liu, M. 1989. A synchronization model for protocol conversion. In *Proceedings of IEEE Infocom 89.* IEEE, New York.

Strom, R. and Yemini, S. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng. SE12,* 1 (Jan.), 157–171.

Strom, R. E., Bacon, D. F., Goldberg, A., Lowry, A., Yellin, D., and Yemini, S. A. 1991. *Hermes: A Language for Distributed Computing.* Prentice-Hall, Englewood Cliffs, N.J.

Strom, R. E. and Yellin, D. M. 1993. Extending typestate checking using conditional liveness analysis. *IEEE Trans. Softw. Eng. SE19,* 5 (May), 478–485.

Thatte, S. 1994. Automated synthesis of interface adaptors for reuseable classes. In *ACM SIGPLAN-SIGACT POPL '94 Conference Proceedings.* ACM, New York, 174–187.

Udell, J. 1994. Componentware. *BYTE 19,* 5 (May).

van den Bos, J. and Laffra, C. 1991. PROCOL - A concurrent object-oriented language with protocols delegation and constraints. *Acta Inf. 28*, 511–538.

Wiederhold, G. 1992. Mediators in the architecture of future information systems. *IEEE Comput. 14,* 3 (Mar.), 38–48.

Received May 1995; revised February 1996; accepted April 1996