

On the construction of components with explicit protocols*

Andrés Farías

Mario Südholt

Département Informatique, École des Mines de Nantes, Nantes, France

<http://www.emn.fr/{farias,sudholt}>

Abstract

Component-based programming promises to facilitate the construction of large-scale applications, which is supported by the important concept of interfaces. In most current component models, interfaces essentially declare types and sets of services that a component implements. They are not expressive enough to formulate many properties important for component collaboration.

In this paper we consider an important class of such properties, sequencing constraints, which components must obey when calling one another services. We consider the integration into interfaces of sequencing properties by means of protocols formalized in terms of finite-state machines. The paper presents three contributions. First, a set of protocol composition operators and a discussion of correctness properties of such operators useful for component assembly. Second, we provide a first step toward the integration of additional state information into protocols. Finally, we show how JavaBeans can benefit from the techniques we present.

Technical Report No: 02/4/INFO

*This work has been partially funded by the EU project “EasyComp” (www.easycomp.org), no. IST-1999-014191

Contents

1	Introduction	3
2	Components with explicit protocols	4
2.1	Component protocols	5
2.2	Properties: protocol substitutability and compatibility	6
3	Composition	7
3.1	Protocol Composition	7
3.1.1	The insertion operator	8
3.1.2	The concatenation operator	9
3.1.3	The union operator	9
3.1.4	Identity constraint propagation operator	11
4	Making JavaBeans' implicit protocols explicit	11
4.1	Basic event management	11
4.2	Bound properties	12
4.3	Constrained properties	12
5	Related Work	15
6	Conclusion	16
A	Theorems and proofs	17
A.1	\uparrow is an equivalence relation	17
A.2	Substitutability of the union protocol operator	18
A.3	Correctness of the moderator protocol	21
A.4	Basic definitions from concurrency theory (for reference)	22

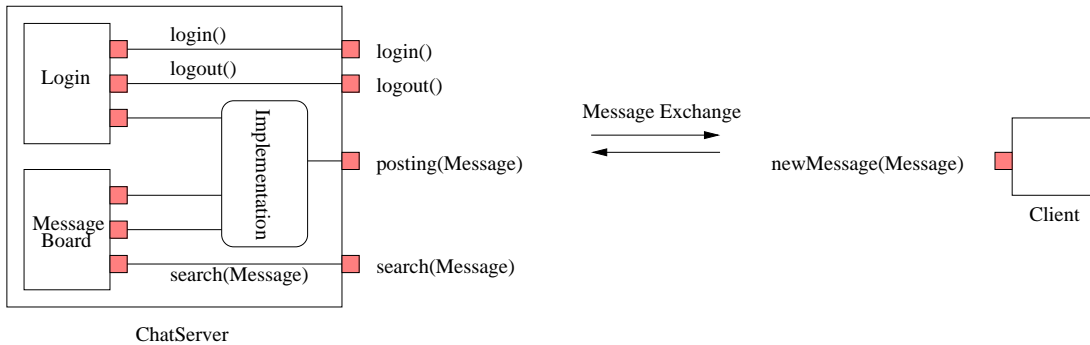


Figure 1: A composition relation where services offered by a component are directly used to define a new component.

1 Introduction

Component-based programming promises to facilitate the construction of large-scale applications by supporting the composition of simple building blocks into complex applications. Despite the huge interest in components within commercial and academic contexts, there is no unique definition of the concept of component. Commonly, several characteristics are accepted as being fundamental to components, in particular explicit interfaces. Interfaces are intended to impose strong restrictions on components: they should make explicit all the means to use components, such as communication, transfer of control between components. This is a much stronger interface notion than is common in object-oriented programming languages where interactions may occur in a hidden fashion, e.g. through a state global to two collaborating objects.

Interfaces of traditional component platforms, such as Sun’s Enterprise JavaBeans [DYK00], define the (Java) type of a component and the types of the services, i.e., methods, provided by a component. More elaborate behavioral specifications are often expressed using separate systems, such as Rational Rose [Kru98, RJB99], State Charts [Har87] for object interactions, and Esterel for synchronization constrains [MF99].¹

An important behavioral property of components are sequencing constraints that components must obey when calling services of another one. Consider the following example that shows the dynamic dependencies regarding service availability in the context of a client-server application. Figure 1 presents a component-based architecture of a chat server application for broadcasting messages among several clients. Components are represented by boxes, and their services by small squares at their border. The `ChatServer` component, for example, offers services for clients to log in and to log out, to broadcast messages to all logged clients and to search a posted message. Component `ChatServer` relays the services of its two collaborators `Login` and `MessageBoard` through its interface, e.g. `login()`, or use them to implement its own services, e.g. `posting()`.

Obviously, the availability of the chat server services depends on its runtime state. Messages, for instance, can only be posted by clients which have previously logged in. The availability of the chat server’s services may not only be dependant on some particular sequence of previously called services but also on other conditions, for example, the identity of components with which the interaction takes place. This is the case of the posting service: it depends on the login service to be called and the identities of the components having logged in.

¹Note that we do not consider ad-hoc techniques, such as the “deployment descriptors” of EJB, which allow to include attributes into interfaces the semantics of which can be arbitrarily complex.

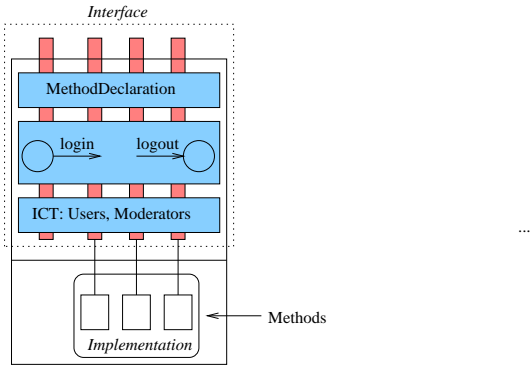


Figure 2: Structure of components with explicit protocols

Subsequent to work on object-oriented languages, explicit protocols in component interfaces have been proposed to facilitate the concise definition of the sequencing constraints of components. Their semantics can be defined using finite-state automata, which support automatic verification techniques and are expressive enough for many application domains.

We build on this work by exploring two enhancements of such techniques. First, we propose several composition operators for explicit protocols and investigate the impact of existing notions of protocol correctness, in particular substitutability, for such operators. These properties are very useful for component-based programming because they can be checked at assembly time. However, we show that they do not hold for some straightforward, i.e. “intuitive”, operator definitions and we propose a technique to solve this problem. Second, we consider the addition of state information to protocols, which restricts protocol transitions based on the identity of collaborating components. Finally, we apply our component model to Sun’s JavaBeans model by making explicit the implicit protocols of JavaBeans components and we exemplify the concise formulation of the assembly of component-based applications using explicit protocols.

The paper is structured as follows: in Section 2, we define our notion of components with explicit protocols. In Section 3, we define the protocol composition operators, discuss their properties and how components are composed. We apply our results to JavaBeans components in Section 4. Related work is discussed in Section 5. Finally, we present a conclusion and propose some future work in Section 6.

2 Components with explicit protocols

We consider components as software units providing an interface consisting of a set of method declarations, one protocol, and a set of lists of identities of collaborating components. The implementation of a component provides implementations for the methods declared in the interface. There are many ways to associate protocols to services, e.g., one protocol per service or one protocol per component. We choose to associate one protocol to a component, because we are interested in expressing strong sequencing constraints. Note that this solution is not less expressive: we can merge, for example, per-service protocols into one protocol using the techniques we present.

Informally, the semantics of interfaces is the following: the method declarations define the services a component offers, the protocol defines sequences of possible interactions (receiving and sending ones) by means of transitions of a finite-state system, and the collaborator lists provide information to restrict protocol transitions based on component identities. Figure 2 illustrates this for the chat component introduced earlier. The provided services include `login()`, the protocol includes a

```

ComponentDefinition ::= component Name Interface Implementation
Interface           ::= MethodDecl* [ProtocolDefinition] ComponentState*
ComponentState    ::= Id : SegList
Implementation    ::= MethodDef*

```

Figure 3: Syntax of a component declarations

```

ProtocolDefinition ::= protocol StateDefinition*
StateDefinition   ::= StateName :: Transition*
StateName        ::= [(init)] Name
Transition       ::= [ComponentTerm:] Direction Name  $\longrightarrow$  Name
ComponentTerm    ::= Name | IdentityConstraintTerm
IdentityConstraintTerm ::= Id+ | Id! | Id- | Id*
Direction       ::= + | -

```

Figure 4: Syntax of a component protocol.

sequencing constraint between login and logout, and a collaborator list records logged-in clients.

In the following we use the syntax shown in Figure 3 to define components.

```

component ChatServer {
  MessageBoard aMessageBoard;
  Login login;
  login(Client aClient);
  logout(Client aClient);
  postMessage(Message m);
  boolean searchMessage(m);

  protocol ChatServer {
    (init) Stable      :: Users+ : +login(Client) --> Stable
                       Users- : +logout(Client) --> Stable
                       -postMessage(Message) --> newMessage
    newMessage :: this: +newMessage() --> Posting
    Posting    :: Users* : +broadcast(Client) --> Stable
  }
}

```

2.1 Component protocols

We consider protocols formalized in terms of finite-state machines. Hence, we define a component protocol as a set of states along with a set of transitions outgoing from each state using the grammar shown in Figure 4.

The initial state of a protocol is identified by the label (*init*). Transitions are labeled with service requests and sequences of such requests are defined as sequences of transitions allowed by the protocol. Transitions are labeled with directed service requests which enable expression of two kinds

```

protocol ChatServer {
  (init) Stable      :: Users+ : +login(Client) --> Stable
                    Users-  : +logout(Client) --> Stable
                    -postMessage(Message) --> newMessage
  newMessage :: this: +newMessage() --> Posting
  Posting    :: Users*  : +broadcast(Client) --> Stable
}

```

Figure 5: Protocol definition of the chat server application.

of service requests: requests by other components to the one considered (direction ‘-’) and requests to ‘the services of other components by the one considered (direction ‘+’).

Transition labels may also include identity constraints, which are lists of component identities. In this case, transitions are triggered only if the corresponding service request is performed by a component whose identity is in the list denoted by the identity constraint term. There are four transition labels concerning such lists:

- $l+$: Add the identity of the component performing the request corresponding to the current transition to list l .
- $l!$: Enable transitions only for components whose identity is in l .
- $l-$: Enable transitions only for components in l and remove the identity of the component requesting the current service.
- $l*$: Sequence of transitions consisting of one transition for each identity in l .

This term can be defined as follows. Let l be a list with n components then $l* : + m()$ is equivalent the following set of transition sequences:

$$\{\langle s_i :: l_{p_i!} + s_{i+1} \rangle_i \mid (p_1, \dots, p_n) \in \text{permutations}(l), i \in \{1, \dots, n\}\}$$

The protocol definitions we introduce can be used to concisely define the interactions of the previous chat server component. Figure 5 shows an appropriate protocol definition. It is defined using three states. The initial state *Stable* has three transitions representing service requests provided by the server (denoted by the direction ‘-’) to add a client, to remove a client and to post a message. The transitions labeled with identity constraints record added and removed clients and thus ensure that messages are posted by clients that are currently logged in to all clients that have been added but not removed.

2.2 Properties: protocol substitutability and compatibility

Explicit protocol information in interfaces, in particular protocols based on finite-state systems, is intended to enable the automatic verification of composition properties and adaptation properties. Existing approaches to explicit protocols for objects and components (see [Nie95, YS97, PV02]) provide a number of suitable properties and verification procedures. Two kinds of property are of foremost importance: compatibility and substitutability. The former is concerned with the problem if traces, i.e. acceptable sequences of service requests, of one protocol can match the traces of another one. In other words whether one component can satisfy the requirements of a second component

for any sequence of service requests. Satisfaction of the latter enables one protocol to be substituted for another. Informally, a protocol replacing another one must accept at least the same sequences of service requests and can not refuse more service requests than the protocol it replaces.

In this paper we consider substitutability properties of the protocol composition operators in some detail and discuss compatibility briefly. To treat substitutability formally, we choose Nierstrasz’ notion of request substitutability [Nie95] and the corresponding notion of substitutability between protocols. (We introduce the relevant notions when they are used later in the text.)

3 Composition

We consider component composition as the basic relation among components. Component composition enables a component to use the services provided by another one. Composition traditionally involves requests to services declared in the interface of another component and modification of values within some data structures for configuration (such as EJB’s “deployment descriptors”). In the context of components with explicit protocols, component composition naturally involves composition of protocols. We propose to support protocol composition by specialized operators. In this paper, we consider four protocol operators:

- *Insertion*. Insertion of a protocol into another one.
- *Append*. Concatenation of a protocol at the end of another one.
- *Union*. Merging several protocols into one.
- *Identity constraint propagation*. Propagation of an identity constraint from one protocol to another one.

(Insertion, Append and Union are structural operators; Identity constraint propagation is an operator for the manipulation of the state associated to protocols.)

These protocol operators then give rise to corresponding component operators as follows. Let a component be denoted as $((d, p, s), i)$ where d is a set of method declarations, p a protocol, s a state associated to a protocol, and i a set of method implementations. Given one of the composition operators introduced above (denoted op), the composition of two components c_1, c_2 can be defined as:

$$((d_1 \cup d_2, p_1 \text{ } op \text{ } p_2, s_1 \cup s_2), i_1 \cup i_2)$$

3.1 Protocol Composition

We formally define these operators using the common definition of finite-state systems as 5-tuples $p = (Q, \Sigma, \delta, i^p, F)$ [HMU01]. Q is the set of states of the component protocol. Σ is the alphabet of valid labels consisting of service requests, i.e. method signatures, directions, and identity constraints. $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, i^p is the initial state and F the set of final states.

For the sake of simplicity, we make two assumptions on protocol definitions. First, operand protocols in a composition are defined using disjoint sets of states. Second, the set of final states of all protocols is $F = \{s \in Q \mid (s, t, s') \in \delta : s = s'\}$; i.e., all states from which no other state can be reached. Similar restrictions are used in related approaches to explicit protocols.²

²Nierstrasz [Nie95], for example, does not mention final states in protocol definitions but considers, for the sake of an argument, all states of a protocol to be final.

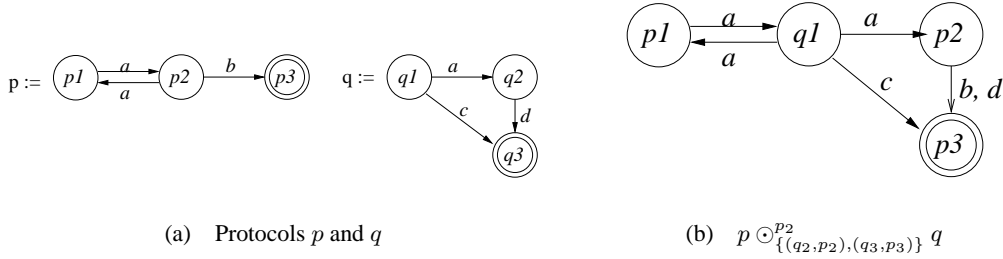


Figure 6: Inserting q into p

Structural manipulation of protocols, i.e., graph structures representing finite-state systems, are difficult to describe using constructive operators, principally because we are interested in preserving properties of protocols, in particular substitutability. We first provide a very general operator for protocol manipulation, the insertion operator, which does not preserve the properties. In a second step, we define three more restricted operators — the append, the union and the identity constraint propagation operators — and discuss property preservation for these operators.

The correctness property we are mainly considering in this paper is protocol substitutability which defines when one protocol can be substituted for another one. We adopt Nierstrasz' notion of request substitutability [Nie95] ensuring that if p is substitutable for q then p can not refuse a service request (after having processed a sequence of service requests, say s) if q could not refuse the same request (also after having already performed s).

In the following, we show that operators for protocols raise a recurring problem with regard to substitutability: operators may introduce non-determinism which invalidates this property. One of the contribution of this paper is a technique to solve this problem in some important cases: we can construct a new protocol which preserves request substitutability by merging specific protocol states. We defer the detailed presentation of this technique up to its application to the union operator below.

3.1.1 The insertion operator

The first operator we consider is very general and allows insertion of a protocol into another one at an arbitrary state, say s , by specifying redirection of transitions to and from s explicitly. For example, protocols p and q shown in Figure 6a. Inserting q in protocol p at state p_2 and replacing states q_2, q_3 with p_2, p_3 , respectively, yields the protocol shown in Figure 6b.

In general, such an insertion depends on four parameters: the two protocols, the target state t where insertion should take place and an identification mapping $I : Q^q \rightarrow Q^p$ which defines states of q that have to be replaced by states from p . The insertion operation can be defined as consisting of two steps. First, every transition starting from t is replaced by an analogous transition starting from q 's initial state (i^q). Second, the identification mapping is processed. For each pair $(s_p, s_q) \in I$, all transitions going to s_q are redirected to s_p and all transitions going out from s_q start at s_p , instead.

Definition: (Insertion operator, \odot) Let p and q be two protocols, $t \in Q^p$, $I : Q^q \rightarrow Q^p$ such that $\exists y \in Q^q : I(y) = t$. The insertion of q into p at t , written $p \odot_I^t q$, is defined by the protocol $r = (Q^r, \Sigma^r, \delta^r, i^p, F^r)$, where:

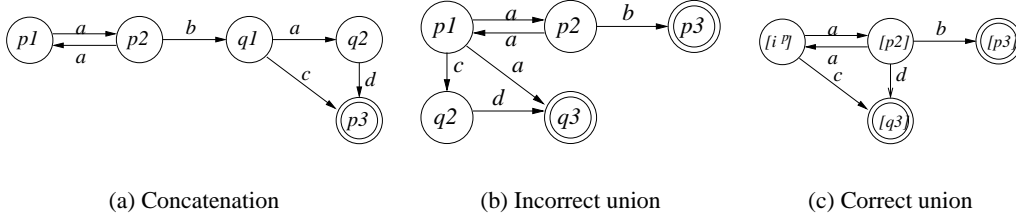


Figure 7: Applying different operators to p and q

$$\begin{aligned}
Q^r &= Q^p \cup Q^q - \text{Dom}(I) \\
\Sigma^r &= \Sigma^p \cup \Sigma^q \\
\delta^r &= \{(x, m, y) \mid (x', m, y') \in (\delta^p \cup \delta^q) : \\
&\quad (x' = I(x) \vee x = x') \wedge (y' = y \vee y' = I(y)) \wedge (y = t \Rightarrow x = t)\} \\
&\quad \cup \{(x, m, i^q) \mid (x, m, t) \in \delta^p, x \neq t\} \cup \{(t, m, t) \in \delta^p\}
\end{aligned}$$

This definition constructs an automaton preserving the states of both automata except for unified states, i.e., states in $\text{Dom}(I)$. In the resulting automaton, the transitions of q et p are preserved if they do not involve states mapped by I and the target state t . Transitions involving states mapped by I are translated to the corresponding transitions involving mapped states. Transitions from other states to t originate from i^q after insertion and self edges on t are preserved. ■

Obviously, the insertion operator does not preserve substitutability property w.r.t. its constituent protocols. This is due to its generality, in particular the structural changes induced by the identification mapping.

3.1.2 The concatenation operator

The concatenation of two protocols consists in appending one protocol to another. The protocol resulting of a concatenation behaves initially as the first constituent protocol but once the protocol reaches any of its final states, it starts behaving as the second protocol. This operator can be defined as a restricted variant of the insertion operator.

Definition: (Concatenation operator \mapsto) Let p and q be two protocols and $f : F^q \rightarrow Q^p, \forall x \in F^q : f(x) = i^p$ an identification mapping. The concatenation of protocol q to protocol p , written $p \mapsto q$, is defined as follows:

$$p \mapsto q := q \odot_f^{i^q} p$$

Figure 7a shows the resulting protocol from concatenating protocol q to protocol p . Regarding protocol properties, the result of a concatenation preserves the properties of its constituents, if restricted to its *right* part. ■

3.1.3 The union operator

Another useful operation for building protocols is a construct that allows any acceptable sequence of any of two protocols. The union is useful, for example, to merge different per-service protocols

into one protocol. A straightforward approach to its definition consists in making the two protocols start from the same initial state but keeping them separate otherwise. However, this operator does not preserve substitutability because the resulting protocol can fail after executing a sequence of service requests when a constituent protocol does not fail. This is due to the non-determinism introduced by merging the initial states of the protocols. Reconsider protocols p and q defined in Figure 6a, the union of which is shown in Figure 7b. The result protocol may fail to execute the sequence $a.b$, while protocol p does not for the same sequence.

In order to preserve substitutability we have to avoid introducing non-deterministic transitions in the resulting protocol if they introduce failures not present in the corresponding constituent protocols. To do this, we have to consider states reached by common sequences of service requests. The problem arises, if the outgoing transitions of the states reached by such a sequence in the two protocols are different. In this case, we propose to merge the concerned states into a new state: every transition ending in one of the old states is redirected to the new one, and transitions starting from one of the old states are made start from the new state. In this way different failures of states reachable by common traces are eliminated.

We can characterize two states problematic in the sense above by means of two relations, denoted \uparrow and $\uparrow\uparrow$. Let p and q be protocols, $x \in Q^p$, $y \in Q^q$, and $i \xrightarrow{s} x$ denote that x is reachable via trace s from i . Then:

$$(i^p, i^q) \stackrel{def}{\in} \uparrow$$

$$x \uparrow y \stackrel{def}{\iff} \exists s : (i^p \xrightarrow{s} x \wedge i^q \xrightarrow{s} y) \wedge \text{initials}(x) \neq \text{initials}(y)$$

where $\text{initials}(x)$ denotes the outgoing transitions of state x . \uparrow relates two states which can be reached by at least one common trace from their respective initial state and which have differently-labeled outgoing transitions.

We denote the reflexive, transitive closure of \uparrow by $\uparrow\uparrow$:

$$x \uparrow\uparrow y \stackrel{def}{\iff} x = y \vee x \uparrow y \vee \exists \langle a_1, \dots, a_n \rangle \subseteq (Q^p \cup Q^q)^n : x \uparrow a_1 \dots a_n \uparrow y$$

The relation $\uparrow\uparrow$ is an equivalence relation (see Theorem 1 in Appendix A.1). We denote its equivalence classes by $[x]^{\uparrow\uparrow}$.

Based on these definition, the union operator can then be defined as follows:

Definition: (Union operator, \oplus) Let p and q be two protocols, the *union operator preserving substitutability*, written $p \oplus q$, is defined as the protocol $r = (Q^r, \Sigma^r, \delta^r, [i^p]^{\uparrow\uparrow}, F^r)$:

$$\begin{aligned} Q^r &= \{[x]^{\uparrow\uparrow} \mid x \in (Q^p \cup Q^q)\} \\ \Sigma^r &= \Sigma^p \cup \Sigma^q \\ \delta^r &= \{(s_1, m, s_2) \mid (x, m, y) \in (\delta^p \cup \delta^q) : s_1 = [x]^{\uparrow\uparrow} \wedge s_2 = [y]^{\uparrow\uparrow}\} \end{aligned}$$

This definition combines two protocols into one by using states merged by means of the relation $\uparrow\uparrow$ (see the use of equivalence classes in the set of states and the definition of the transition relation). This definition of the union operator ensures that the resulting protocol preserve the substitutability relation w.r.t. its operands (see Theorem 2 in Appendix A.2). ■

Applied to the two example protocols shown in Figure 6a, the union operator yields the result shown in Figure 7c.

3.1.4 Identity constraint propagation operator

The operators defined previously are structural operators because they are defined only in terms of states and transitions representing service requests. Our protocols also include state information to record identities of collaborating components and restrict transitions based on identities. Protocols can therefore be combined in order to manipulate this state information. In this section we present such an operator that propagates identity constraints among protocols.

One operator for identity constraint propagation may label all its transitions with a identity constraints. This operator can be defined as follows:

Definition: (Identity constraint propagation operator, \downarrow) Let p be a protocol and X an identity constraint term. The *identity constraint propagation* constraining p by X , written $p \downarrow_X$, is defined as the protocol $r = (Q^r, \Sigma^r, \delta^r, i^r, F^r)$ where:

$$\begin{aligned}\Sigma^r &= \{Z : \pm m \mid Z = (Y \cap X), Y : \pm m \in \Sigma^p\} \\ \delta^r &= \{(x, Z : \pm m, y) \mid Z = (Y \cap X), (x, Y : \pm m, y) \in \delta^p\}\end{aligned}$$

■

This operator does not preserve substitutability by itself but there are state-specific laws guaranteeing the preservation of substitutability for parts of the result, for example, where $Y \subseteq X$.

This operation can be combined with the other operators. A specialized concatenation operator of protocols, for example, may propagate an identity constraint from final transitions of the first protocol to the second one in addition to concatenation. More precisely, let p and q be two protocols such that every transition leading to a final state of p is constrained by the same component term X . The result of propagating the constraint X of p to q in $p \mapsto^X q$ can be defined as:

$$r = (p \mapsto q) \downarrow_X$$

4 Making JavaBeans' implicit protocols explicit

JavaBeans [Ham97] is a white-box component model based on Java [Sun95]. The communication means of a JavaBean consists of public fields, methods, and several mechanisms based on events. The mechanisms for event-based communication can be seen as defining implicit protocols between communicating JavaBeans. The protocols are implicit because parts of them do not appear in the JavaBeans' interface. In this section, we make these protocols explicit using the notions introduced previously and we show how we can define JavaBeans more declaratively.

JavaBeans' events adhere to the publish-subscribe paradigm [EGD01], permitting components to register for notification (through broadcast) of events. Basically, two implicit protocols are supported: the *bound properties* represent values for which the modification is notified through a broadcast event to all registered components. *Constrained properties* are values for which the modification any registered component has the possibility to emit a veto to a proposed modification. These particular kinds of properties are used as a composition mechanism, specially in GUI interfaces, where, for instance, buttons and other graphics elements are plugged together in order to conform a new interface entity.

4.1 Basic event management

Events are used to propagate information from a source bean to a set of collaborating beans, called listeners. Event objects, of type `EventObject` encapsulate information about a state change of an *event source* bean. The listener beans must implement an interface inheriting from `EventListener`

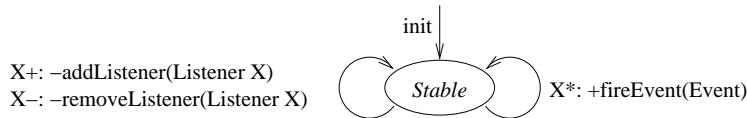


Figure 8: Protocol for basic event management

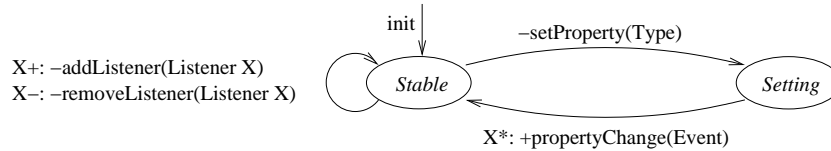


Figure 9: Protocol of bound properties

in order to being notified about the change of the value of an event source. The event source bean must implement two methods for adding and removing listeners.

The event registering mechanism essentially is a (simple) protocol between an event source and one or more event listeners. We can represent this protocol explicitly using our notation as shown in Figure 8. The protocol is defined by one state with three transitions constrained by identity constraints. The transition labeled with `addListener(...)` allows any component to call the service for subscribing as event listener and to add its identity to the variable `X`. The transition labeled `removeListener(...)` allows a component to request its removal from the list of listeners provided that it has been previously registered. Finally, the transition labeled `fireEvent(...)` describes the source sending an event to all registered listeners.

4.2 Bound properties

A *bound property* is an instance variable of a JavaBean satisfying two characteristics. First, other beans can access a bound property only through its accessors methods for getting and setting its value. Second, the beans owning the bound property keeps a list of subscribed listeners that are notified with a suitable event each time that the bound variable changes.

This protocol can be represented explicitly as shown in Figure 9. Basically, bound properties reuse the protocol for basic event management and add one state which is reached by setting the property. Once the property has been set, the protocol transits to its initial state by broadcasting an appropriate event to all listeners.

4.3 Constrained properties

In the case of constrained properties, the source bean keeps, in general, two lists of listeners: one list of beans listening to changes of the constrained property and another list of beans that can veto a change to the property's value. Once a change has been requested, the bean broadcasts an event to the beans in the second list which can veto or not the proposed change. If a bean disagrees with the change, it throws an exception of type `PropertyVetoException`. Otherwise, the value is changed and a `changeProperty` event is broadcast to listeners registered to be notified.

The protocol of constrained properties can be made explicit as shown in Figure 10. Compared to the protocol for bound properties, this protocol features an intermediate state *Vetoing* that allows beans to emit a veto, in which case no change occurs. Otherwise the protocol transits to the *Setting*

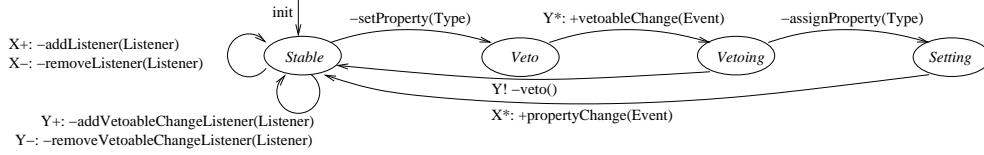


Figure 10: Component protocol of a constrained property.

state from where the bean fires a `changeProperty` event to every listener registered for change notification.

This discussion suggests that constrained properties can be seen as an extension of bound properties. Since it is possible to express these protocols explicitly in our framework, we can define the relationship between the two protocols precisely. The following protocol definition captures the veto-part of the protocol for the constraint property

```

protocol Veto {
  Veto    :: Y*      : +vetoableChange(Event) --> Vetoing
  Vetoing :: +assignProperty(Type)           --> Approved
          Y!      : -veto()                  --> (init) Start
}

```

In state `Veto` the bean notifies all beans registered in `Y` of the change proposal. Then, in state `Vetoing`, two transitions can be taken. Either an `assignProperty` message is sent to every component in `Y` and the `Approved` state is reached. Or the change is vetoed by one of the listeners which is represented by the message `veto`.

The constrained protocol can then be written as a protocol composition between the bound protocol and the `Veto` protocol as follows

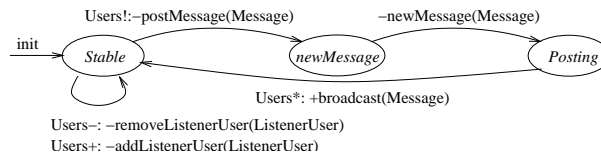
(The proof of this equality is given in Theorem 7 in Appendix A.3):

$$\text{ConstrainedProtocol} = \text{BoundProtocol} \odot_{\{(Approved, Setting), (Vetoed, Stable)\}}^{\text{Setting}} \text{Veto}$$

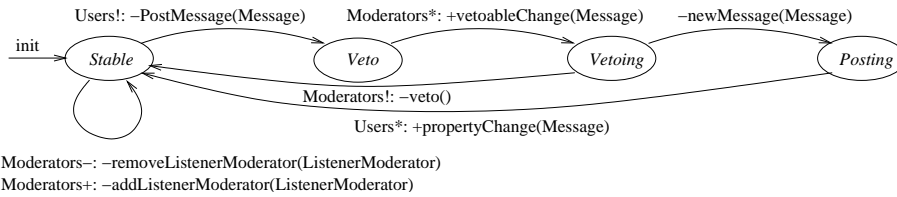
To demonstrate the application of the proposed formalism, consider the use of bound and constrained protocols as part of the chat server application. Sometimes, a special chat session may require one or more moderators that may refuse the broadcasting of an inadequate message. This can be implemented by constraining the messages to be sent with the vetos of moderators. Figure 11 shows an appropriate protocol. Part (a) shows the protocol for posting (broadcasting) messages. From an initial state, it is possible to either add or remove users that can request to post a message that is afterward broadcast to every registered user. Part (b) shows the protocol for moderation of messages using a constrained property. From an initial state, moderators can be added or removed and their approval is requested whenever a message is posted. If no moderator disagrees the message is posted.

The moderated version of the posting protocol can be composed from protocols a) and b) as shown in Part c). The protocol composition essentially consists in inserting the moderator protocol into the posting protocol by binding their initial and posting states. Let $l = \{(Stable_M, Stable_P), (Posting_M, Posting_P)\}$ be an identification mapping. The complete protocol can then be defined as a composition:

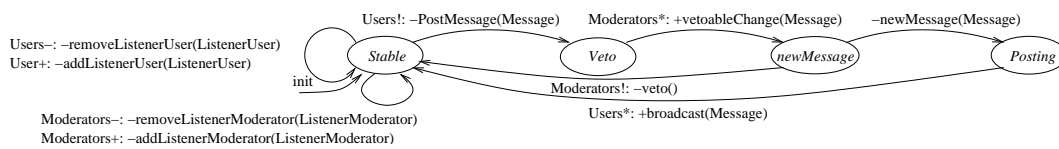
$$\text{ModeratedPosting} = \text{Posting} \odot_l^{\text{Stable}} \text{Moderator}$$



(a) The posting protocol.



(b) The moderator protocol.



(c) The resulting protocol.

Figure 11: Moderators protocol.

```

protocol Login {
  Stable :: X+ : +login() --> Stable
}

```

Figure 12: Login and Logout protocols.

A chat server protocol, which provides a monitored posting service and a message board search facility, can then be defined by the following expression:

$$Chat = ModeratedPosting \oplus SearchMessage$$

The protocol `Chat` is substitutable for both protocols and can therefore safely replace any server providing only one of these. Finally, we can require clients and moderators to log in first using the protocol shown in Figure 12 as follows:

$$SecureChat = Login \mapsto^X Chat$$

These examples show that our operators can be used to define certain common compositions quite concisely and declaratively.

5 Related Work

There are many approaches to the specification of sequencing constraints in many different fields. In the following we only consider approaches concerned with explicit protocol specifications based on finite-automata in components or object-oriented systems.

Objects and protocols. Nierstrasz uses regular types [Nie95] to investigate service availability of objects using CSP [BHR84] as a basis. He defines notions for compatibility and substitutability of protocol-enhanced objects. Our work can be seen as an extension of his work to components. PROCOL [VL89] is a parallel C-based object-oriented language with explicit per-objects protocols. Protocols describe the sequencing and synchronization constraints between an object and its partners. Protocols can constrain access to certain methods directly referencing instance variables by means of guards. PROCOL is intended to be a general programming language and does not provide support for the automatic verification of properties. In contrast to our work, neither of these approaches considers construction operators and state separated from the protocol.

Component and protocols. There are several approaches to the integration of finite-state based protocols to components. Plazil et al. from the SOFA project [SOF] at Charles University in Prague propose an enhanced architectural description language for component behavior with explicit protocols. They investigate protocol composition operators similar to regular expressions [PV02]. However, they do not consider property preservation of such operators. Yellin and Strom [YS97] also integrate explicit protocol into components. Their work is genuine in that it considers automatic generation of adapter code among component protocols in order to satisfy compatibility. However, they do not consider constructors operators. Alfaro and Henzinger [dAH01] also explore temporal properties of components by means of finite-state machines. They do consider construction operators but use an “optimistic” semantics for protocols. This means that their correctness properties are very different from ours. Finally, these three approaches do not consider separated state information associated to protocols.

Separated component specifications. There are numerous approaches supporting specifications of component-related properties which are separated from the underlying component model itself. UML [RJB99], for instance, has been applied to the specification of components. Similarly, message sequence charts [RGG96] (for an application to components see the PhD thesis of Wydaeghe [Wyd01]) is a trace language which can be used to describe interactions among components based on finite-state systems. Architectural description languages, such as Wright [All97], have also been used for similar purposes. In contrast to our work, few of these approaches have considered construction operators and none of them explores the separation of state information from the protocol.

6 Conclusion

Extending previous work on finite-state based protocols for objects and components, we proposed two main contributions. First, a set of four composition operators for components with explicit protocols and discussed the impact of existing notions of protocol correctness, in particular substitutability. We showed, in particular, that they do not hold for some straightforward, i.e. “intuitive”, operator definitions and we presented a technique to solve this problem. Second, we considered the addition of state information to protocols in order to restrict protocol transitions based on the identity of collaborating components. We validated the use of the proposed techniques by applying them to Sun’s JavaBeans model. We have been able to make explicit the implicit protocols of JavaBeans components and demonstrated that explicit protocols support a concise method for the declarative assembly of component-based applications.

Future work. There are many topics for future work based on the work presented in this paper. The current set of protocol constructors should be enlarged in order to make protocol construction more flexible. In particular, the gap between the very general insertion operator (which does not preserve properties) and the other rather restricted operators (which do preserve properties) should be bridged. Furthermore, our investigation of separated state information only constitutes a first step of this concept and should be pursued.

A Theorems and proofs

Remark: The proofs in this section make use of four basic notions of concurrency theory: initials, traces, failures and relative failures. Their definitions according to Nierstrasz [Nie95] can be found in Appendix A.4.

A.1 \uparrow is an equivalence relation

Theorem 1 *The relation \uparrow , defined as follows (reprinted here from page 10):*

$$(i^p, i^q) \stackrel{def}{\in} \uparrow$$

$$x \uparrow y \stackrel{def}{\iff} \exists s : (i^p \xrightarrow{s} x \wedge i^q \xrightarrow{s} y) \wedge \text{initials}(x) \neq \text{initials}(y) \quad (1)$$

$$x \uparrow y \stackrel{def}{\iff} x = y \vee x \uparrow y \vee \exists \langle a_1, \dots, a_n \rangle \subseteq (Q^p \cup Q^q)^n : x \uparrow a_1 \dots a_n \uparrow y \quad (2)$$

is an equivalence relation.

Proof: An equivalence relation is reflexive, symmetric and transitive.

1. Reflexivity: $x \uparrow x$ trivially holds by the first conjunctive term in Equation 2.
2. Symmetry: Suppose that $x \uparrow y$. We proof that all three disjunctive terms in Equation 2 are symmetric. If $x = y$ then automatically $y \uparrow x$. the relation \uparrow (see Equation 1) is symmetric because the two conditions on x and y , namely existence of trace s and difference of initial sets (see Definition 1), are symmetric. Then if $x \uparrow y$ then $y \uparrow x$. In the third case, x and y satisfy that

$$\exists \langle a_1, \dots, a_n \rangle \subseteq (Q^p \cup Q^q)^n : x \uparrow a_1 \dots a_n \uparrow y$$

Because \uparrow is symmetric, we have that

$$\langle a_n, \dots, a_1 \rangle \subseteq (Q^p \cup Q^q)^n : y \uparrow a_n \dots a_1 \uparrow x$$

which implies that $y \uparrow x$.

3. Transitivity: If $x \uparrow y$ and $y \uparrow z$, then:

$$\exists \langle a_1, \dots, a_n \rangle \subseteq (Q^p \cup Q^q)^n : x \uparrow a_1 \dots a_n \uparrow y$$

and

$$\exists \langle b_1, \dots, b_n \rangle \subseteq (Q^q \cup Q^p)^n : y \uparrow b_1 \dots b_n \uparrow z$$

then

$$\langle a_1, \dots, a_n, b_1, \dots, b_n \rangle \subseteq (Q^q \cup Q^p)^{2n} : x \uparrow a_1 \dots a_n \uparrow y \uparrow b_1 \dots b_n \uparrow z$$

so $x \uparrow z$.

■

A.2 Substitutability of the union protocol operator

The proof that the resulting protocol of a union operation can substitute any of its operands mainly relies on the fact that the result protocol inherits all the traces from its operands. Moreover, the states of the result protocol r are created in such a way that the set of failures of r cannot be larger than the failure sets of its operands.

In the following, we denote $failures^p(x)$ the failures of the state x at protocol p and $failures_s^p(i_p)$ the failures of protocol p at state i^p relative to state s (see Definitions 3 and 4 in Appendix A.4).

Theorem 2 *Let $r = p \oplus q$. Then r is request substitutable for protocols p and q .*

Proof:

Without loss of generality we prove that r is request substitutable for p (substitutability for q follows from \oplus being commutative (see Lemma 3). Applying Nierstrasz' notion of protocol substitutability, protocol r can substitute protocol p iff:

$$traces(i^p) \subseteq traces(i^r) \quad (3)$$

$$failures_{i^p}(i^r) \subseteq failures(i^p) \quad (4)$$

We will prove that protocol r at its initial state is request substitutable for protocol p at its initial state.

Property 3 follows from Lemma 5.

For Property 4 we must prove that if $(t, R) \in failures_{i^p}(i^r)$ then $(t, R) \in failures(i^p)$.

Let $(t, R) \in failures_{i^p}(i^r)$, then there exists a state $s \in Q^r$ such that $i^r \xrightarrow{t} s$ and $R \cap initials(s) = \emptyset$, with $R \subseteq \Sigma^r - initials(s)$. Let $P_t \stackrel{def}{=} \{x \in Q^p \mid s = [x]^\uparrow \wedge i^p \xrightarrow{t} x\}$. As proven in Lemma 6 any message accepted in state x such that $s = [x]$ is also accepted in state s . The same is true for the elements in P_t . More formally:

$$initials(s) = \bigcup_{z \in X_s} initials(z) \cup \bigcup_{z \in Y_s} initials(z)$$

\Rightarrow

$$initials(s) \supseteq \bigcup_{z \in X_s} initials(z) \supseteq \bigcup_{z \in P_t \subseteq X_s} initials(z)$$

The last expression means that any accepted message by a state x in p is also accepted in s . In other words, any refused message belonging to a failure (t, R) will necessarily be refused at every state that has been merged to form the state s . ■

Lemma 3 \oplus is commutative.

Proof:

Let $r = p \oplus q$. Then, by Definition 3.1.3, $r = (Q^r, \Sigma^r, \delta^r, i^p, F^r)$ where:

$$Q^r = \{[x]^\uparrow \mid x \in (Q^p \cup Q^q)\} \quad (5)$$

$$\Sigma^r = \Sigma^p \cup \Sigma^q \quad (6)$$

$$\delta^r = \{(s_1, m, s_2) \mid (x, m, y) \in (\delta^p \cup \delta^q) : s_1 = [x]^\uparrow \wedge s_2 = [y]^\uparrow\} \quad (7)$$

All the expressions used in Equations 5, 6 and 7 are commutative, and therefore \oplus is commutative as well. ■

The following two lemmas shows that all the traces defined in the operand protocols also belong to the protocol resulting from their union.³

Lemma 4 *Let $r = p \oplus q$. If $i^p \xrightarrow{t} s$ or $i^q \xrightarrow{t} s$ then $i^r \xrightarrow{t} s'$ such that $s' = [s]^\uparrow$.*

Proof:

Without loss of generality we prove the property for traces in protocol p . We prove it by induction over the length of the transition $t \in (\delta^r)^*$.

$|t| = 1$. If trace t has length 1 and $i^p \xrightarrow{t} s$ then there is a transition $(i^p, t, s) \in \delta^p$. By Definition 7, transition (i^p, t, s) is replaced by transition (i^r, t, s') such that $s' = [s]^\uparrow$.

$|t| = (n + 1)$. Let t be a trace of size $n + 1$ such that $i^p \xrightarrow{t} s$. Trace t can be written as $t = t' \cdot m$. Because $t' \in \text{traces}_p$ and the induction hypothesis there is a non empty set $P_t = \{s \in Q^p \mid i^p \xrightarrow{t'} s\}$. Let $s' \in P_t$ such that $(s', m, s) \in \delta^p$. By induction hypothesis, given that $i^p \xrightarrow{t'} s'$ then there is a transition $i^r \xrightarrow{t'} s''$, $s'' \in Q^r$ such that $s'' = [s']^\uparrow$. Therefore, transition $(s', m, s) \in \delta^p$ with $s'' = [s']^\uparrow$, generates, by construction (see Equation 7), a transition $(s'', m, s''') \in \delta^r$ such that $s''' = [s]^\uparrow$.

■

Lemma 5 *Let $r = p \oplus q$. Then, every trace in the traces of p or q belongs to the traces of r , too. More formally:*

$$\text{traces}_p \cup \text{traces}_q \subseteq \text{traces}_r$$

Proof:

Let $t \in \text{traces}_p \cup \text{traces}_q$, then $t \in \text{traces}_p \vee t \in \text{traces}_q$. Without loss of generality, we suppose $t \in \text{traces}_p$ and prove $t \in \text{traces}_r$. If $t \in \text{traces}_p$, there is a state $s \in Q^p$ such that $i^p \xrightarrow{t} s$. By lemma 4, $i^r \xrightarrow{t} s'$ such that $s' = [s]$. Therefore $t \in \text{traces}_r$.

■

Finally, for the proof of the failure condition in Theorem 2, we have to proof that the result protocol of a union operation preserves the initials of its operand protocols.

Lemma 6 *Let $r = p \oplus q$. Let $s \in Q^r$, $X_s \subseteq Q^p$ and $Y_s \subseteq Q^q$ such that for all $z \in (X_s \cup Y_s) : s = [z]^\uparrow$, then:*

$$\text{initials}^r(s) = \bigcup_{z \in X_s} \text{initials}^p(z) \cup \bigcup_{z \in Y_s} \text{initials}^q(z)$$

Proof:

$\forall x, y, z$ it must be proven that

$$((x, m, s_1) \in \delta^p \Rightarrow (z, m, s_2) \in \delta^r) \wedge ((y, m, s_1) \in \delta^q \Rightarrow (z', m, s_2) \in \delta^r) \quad (8)$$

³The inverse property also holds, i.e., every trace in the resulting protocol belongs to at least one of its operand protocols. This means that in the process of a union, traces are neither created nor lost. (However, this stronger property is not necessary for the proofs we are interested in.

and

$$(z, m, s_2) \in \delta^r \Rightarrow (x, m, s_1) \in \delta^p \vee (y, m, s'_1) \in \delta^q \quad (9)$$

such that $z = [x]^\uparrow$ and $z = [y]^\uparrow$.

Equation 8. Let $(x, m, s_1) \in \delta^p$ and $z = [x]^\uparrow$. Then, by construction, $(z, m, s_2) \in \delta^r$ such that $z = [x]^\uparrow$ and $s_2 = [s_1]^\uparrow$. Let $(y, m, s_1) \in \delta^q$ and $z = [y]^\uparrow$. Then, by construction, $(z, m, s_2) \in \delta^r$ such that $z = [y]^\uparrow$ and $s_2 = [s_1]^\uparrow$.

Equation 9. Let $(z, m, s_2) \in \delta^r$. By definition, if $(z, m, s_2) \in \delta^r$ then there is $(x, m, s_1) \in (\delta^p \cup \delta^q)$ such that $z = [x]^\uparrow$ and $s_2 = [s_1]^\uparrow$.

■

A.3 Correctness of the moderator protocol

Theorem 7 *The following equation holds:*

$$\text{ConstrainedProtocol} = \text{BoundProtocol} \odot_{\{(Approved, Setting), (Vetoed, Stable)\}}^{\text{Setting}} \text{Veto}$$

Proof:

Denoting terms of the protocol *BoundProtocol* using b and protocol *Veto* as v , we can prove the equation by substituting equals for equals. Let $b_1 \dots b_4$ and $v_1 \dots v_3$ denote the following which are part of the two protocols:

$$\begin{aligned} b_1 &= X+ : -\text{addPropertyListener}(\text{Listener}) \\ b_2 &= X- : -\text{removeListener}(\text{Listener}) \\ b_3 &= -\text{setProperty}(\text{Type}) \\ b_4 &= X* : +\text{propertyChange}(\text{Event}) \\ \\ v_1 &= Y* : -\text{vetoableChange}(\text{Event}) \\ v_2 &= -\text{assignProperty}(\text{Type}) \\ v_3 &= Y! : +\text{veto}() \end{aligned}$$

The constrained protocol is defined as the protocol $r = (Q^r, \Sigma^r, \delta^r, i^b, F^r)$, where

$$\begin{aligned} Q^r &= Q^b \cup Q^v - \text{Dom}(I) \\ \Sigma^r &= \Sigma^b \cup \Sigma^v \\ \delta^r &= \{(x, m, y) \mid (x', m, y') \in (\delta^b \cup \delta^v) : \\ &\quad (x' = I(x) \vee x = x') \wedge (y' = y \vee y' = I(y)) \wedge (y = t \Rightarrow x \neq t)\} \\ &\quad \cup \{(x, m, i^v) \mid (x, m, t) \in \delta^b, x = t\} \cup \{(t, m, t) \in \delta^b\} \end{aligned}$$

replacing the values we obtain:

$$\begin{aligned} Q^r &= \{\text{Stable}, \text{Setting}\} \cup \{\text{Veto}, \text{Vetoing}, \text{Approved}, \text{Vetoed}\} - \{\text{Approved}, \text{Vetoed}\} \\ \Sigma^r &= \{b_1, b_2, b_3, b_4\} \cup \{v_1, v_2, v_3\} \\ \delta^r &= \{(\text{Stable}, b_1, \text{Stable}), (\text{Stable}, b_2, \text{Stable}), (\text{Setting}, b_4, \text{Stable}), (\text{Veto}, v_1, \text{Vetoing}), \\ &\quad (\text{Vetoing}, v_2, \text{Setting}), (\text{Vetoing}, v_1, \text{Stable})\} \cup \{(\text{Stable}, b_3, \text{Veto})\} \cup \{\} \end{aligned}$$

Reducing this protocol definition, we obtain:

$$\begin{aligned} Q^r &= \{\text{Stable}, \text{Setting}, \text{Veto}, \text{Vetoing}\} \\ \Sigma^r &= \{b_1, b_2, b_3, b_4, v_1, v_2, v_3\} \\ \delta^r &= \{(\text{Stable}, b_1, \text{Stable}), (\text{Stable}, b_2, \text{Stable}), (\text{Setting}, b_4, \text{Stable}), (\text{Veto}, v_1, \text{Vetoing}), \\ &\quad (\text{Vetoing}, v_2, \text{Setting}), (\text{Vetoing}, v_1, \text{Stable}), (\text{Stable}, b_3, \text{Veto})\} \end{aligned}$$

which corresponds to the constrained protocol shown in Figure 10. ■

A.4 Basic definitions from concurrency theory (for reference)

The following definitions are repeated here from Nierstrasz [Nie95].

The *initials* of a state s of a protocol p are defined as the set of services which can be called in state s .

Definition 1

$$initials(s) = \{t \mid \exists s' \in Q^p, (s, t, s') \in \delta^p\}$$

■

The *traces* of a protocol p from a state s are the set of valid sequences of service calls starting in s .

Definition 2

$$traces(s) = \{t \in (\delta^p)^+ \mid \exists s' \in Q^p, s \xrightarrow{t} s'\}$$

■

Given a protocol p , the set of *failures* of a state s is the set of pairs (t, R) such that p may accept trace t starting in s but then refuse any of the service requests occurring in R .

Definition 3

$$failures(s) = \{(t, R) \mid \exists s', s \xrightarrow{t} s', R \text{ is finite, } R \cap initials(s') = \emptyset\}$$

■

The *relative failures* of s w.r.t. to another state t (normally belonging to another protocol than s) are the failures of s generated by accepting valid traces starting in t .

Definition 4

$$failures_t(s) = \{(u, R) \in failures(s) \mid u \in traces(t)\}$$

■

References

- [All97] Robert J. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.
- [BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, July 1984.
- [dAH01] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proc. of the 8th European Software Engineering Conference and 9th Symposium on the Foundation of Software Engineering (ESEC/FSE)*, volume 26, 5 of *Software Engineering Notes*. ACM, September 10–14 2001.
- [DYK00] Linda DeMichiel, Ümit Yalçinalp, and Sanjeev Krishnam. Enterprise JavaBeans specification, version 2.0. Technical report, SunMicrosystems, October 2000.
- [EGD01] P. Eugster, R. Guerraoui, and C. Damm. On objects and events. In *Proceedings OOPSLA*, October 2001.
- [Ham97] Graham Hamilton. Java beans API specification. Technical report, Sun Microsystems, July 1997.
- [Har87] David Harel. Statecharts: A visual formalism for complex system. *Science of Computer Programming*, 8(3):231–274, March 1987.
- [HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2001.
- [Kru98] Philippe Kruchten. *Rational Unified Process: an Introduction*. Addison-Wesley, Reading, Massachusetts, USA, 1998.
- [MF99] Frédéric Mallet and F.Boéri. Esterel and java in an object-oriented framework for heterogeneous software and hardware system modelling and simulation. the sep approach. In *Euromicro Conference*, volume I, pages 214–222, Milan, Italie, September 1999.
- [Nie95] Oscar Nierstrasz. Regular types for active objects. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.
- [PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. In *Transactions on Software Engineering*. IEEE, January 2002.
- [RGG96] Ekkart Rudolph, Peter Graubmann, and Jens Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, June 1996.
- [RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, USA, 1 edition, 1999.
- [SOF] SOFA project. <http://nenya.ms.mff.cuni.cz/thegroup/SOFA/sofa.html>.
- [Sun95] Sun Microsystems. The Java language: An overview. Technical report, Sun Microsystems, 1995.

- [VL89] Jan Van Den Bos and Chris Laffra. PROCOL: a parallel object language with protocols. *ACM SIGPLAN Notices*, 24(10):95–102, October 1989.
- [Wyd01] Bart Wydaeghe. *PACOSUITE, Component Composition Based on Composition Patterns and Usage Scenarios*. PhD thesis, Vrije Universiteit Brussels, 2001.
- [YS97] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions of Programming Languages and Systems*, 19(2):292 – 333, March 1997.