

On components with explicit protocols satisfying a notion of correctness by construction^{*}

Andrés Fariás and Mario Südholt

Département Informatique
École des Mines de Nantes
4, rue Alfred Kastler – BP 20722
F-44307 Nantes Cedex 3, France
<http://www.emn.fr/{farias,sudholt}>

Abstract. Component-based programming promises to facilitate the construction of large-scale applications, it is supported by the concept of interfaces. In most current component models, interfaces essentially declare types and sets of services that components implement. However, interfaces are not expressive enough to formulate structural and behavioral properties important for component collaboration. We consider an important class of component collaboration properties: sequencing constraints, which components must obey when calling one another's services.

In this paper, we integrate sequencing properties into interfaces by means of protocols formalized in terms of finite-state machines. We are interested in operators for the construction of components, which satisfy a correctness property which allows a component to be substituted by another one. We present three main contributions. First, we define a set of protocol composition operators, which allow protocols to be constructed while ensuring substitutability. Second, we provide a first step toward the integration of additional abstract state information into protocols. Finally, we show that the model presented can be advantageously applied to the definition and extension of two widely-used component models: JavaBeans and EJB. We show how to make the JavaBeans' implicit protocols explicit and extend the formulation of EJB's access-control policies at the interface level.

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 and transfer of control between components. This is a much stronger notion of interface than is common in object-oriented programming languages, where interactions may

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

occur in a hidden fashion, e.g. through a state global to two collaborating objects (field, constant pool).

Interfaces of traditional component platforms, such as Sun’s Enterprise JavaBeans (EJB) [1], 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 methodologies, such as Rational Rose [2, 3], State Charts [4] for object interactions, and Esterel for synchronization constraints [5]. In the case of EJB, concerns such as security and persistence, are declaratively configured by means of *deployment descriptors*, which are files intended to separate such behavioral descriptions from the code of the enterprise beans. This way an EJB can be configured by the bean deployer without intervening on the code.

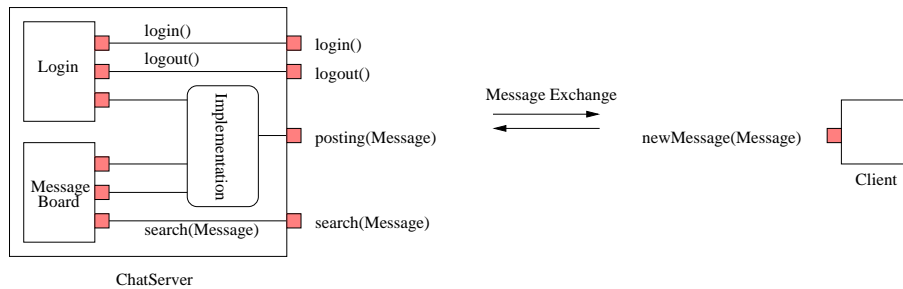


Fig. 1. A composition relation where services offered by a component are directly used to define a new component.

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 for 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 and eventually on the state of its clients or collaborators. 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 depend 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.

Subsequent to work on object-oriented languages, explicit protocols in component interfaces have been proposed by [6, 7, 8, 9] to facilitate the concise definition of 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 define component composition operators for explicit protocols and investigate the impact of existing notions of protocol correctness, in particular substitutability, for such operators. These properties are capital for component-based programming because they allow applications to be extended without modifying their clients or avoiding the construction of dynamic adaptors, which is a hard task and costly. Properties such as substitutability 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 according to the identity of collaborating components. Third, 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. We use the example of the chat server to show how this can be done in JavaBeans. Finally, we show how easily protocols can be used to define access-control security policies that cannot be defined in the EJB's security model.

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. We show how to extend the EJB's security model in Section 5. Related work is discussed in Section 6. Finally, we present a conclusion and propose some future work in Section 7.

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 that are used by the protocol to control the reception and send of methods calls. 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.

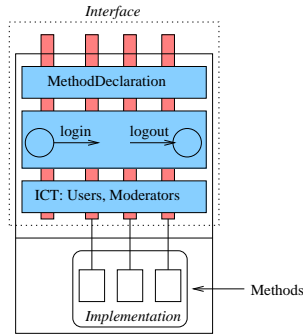


Fig. 2. Structure of components with explicit protocols

The provided services include `login()`, the protocol includes a sequencing constraint between `login` and `logout`, and a collaborator list records logged-in clients.

$$\begin{aligned}
 \textit{ComponentDefinition} & ::= \textit{component Name Interface Implementation} \\
 \textit{Interface} & ::= \textit{MethodDecl}^* [\textit{ProtocolDefinition}] \textit{ComponentState}^* \\
 \textit{ComponentState} & ::= \textit{Name} : \textit{IdentityList} \\
 \textit{Implementation} & ::= \textit{MethodDef}^*
 \end{aligned}$$

Fig. 3. Syntax of a component declaration

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

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.

$$\begin{aligned}
 \textit{ProtocolDefinition} & ::= \textit{protocol StateDefinition}^* \\
 \textit{StateDefinition} & ::= \textit{StateName} :: \textit{Transition}^* \\
 \textit{StateName} & ::= [(\textit{init})] \textit{Name} \\
 \textit{Transition} & ::= [\textit{ComponentTerm}.] \textit{Direction Name} \longrightarrow \textit{Name} \\
 \textit{ComponentTerm} & ::= \textit{Name} \mid \textit{IdentityConstraintTerm} \\
 \textit{IdentityConstraintTerm} & ::= \textit{Id}+ \mid \textit{Id}! \mid \textit{Id}- \mid \textit{Id}^* \\
 \textit{Direction} & ::= - \mid +
 \end{aligned}$$

Fig. 4. Syntax of a component protocol.

The initial state of a protocol is identified by the label (`init`). Transitions are labelled with service requests and sequences of such requests are defined as sequences of transitions allowed by the protocol. Transitions are labelled with directed service requests which enable expression of two kinds 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 to the following set of transition sequences:

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

```

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

```

Fig. 5. Protocol definition of the chat server application.

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 labelled 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. Once a client has successfully requested to post a message the protocol transits to the *newMessage* state from which the only allowed operation allows the server to set a *newMessage* and pass to the *Posting* state. The protocol can transit to the *Stable* once it has sent a *broadcast(Message)* message to every client that has been logged. This protocol ensures that while a message is being posted and broadcasted no other clients can logout or login.

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 [10, 6, 9]) 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 one another. 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. Moreover, if a protocol q is substitutable for protocol p , then protocol q is compatible with every protocol compatible with p . In other words, if we know that a component server is compatible with a set of clients, then any component with a protocol that can substitute the original protocol of the server will still be compatible with the clients.

We strongly believe that substitutability is a key property in the development of distributed applications because it allows one to determine, for example, whether an extended server is still compatible with its clients. 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 [10] and the corresponding notion of substitutability between protocols. (We introduce the relevant notions as needed in the text.)

3 Composition

We consider component composition as the basic relation that enables a component to use the services provided by another one. Composition traditionally involves requests to services declared in the interface of another component. In the context of components with explicit protocols, component composition naturally involves composition of protocols. We propose to support protocol composition by protocol composition operators because we are interested in — as far as possible — preserving the substitutability by construction. In this paper, we consider five protocol operators:

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

While the first four operators, Union at state, Union, Append and Insertion are structural operators, the last operator, Identity constraint propagation, is an operator for the manipulation of the abstract state associated with protocols. The interest of these first

four operators is to permit the construction of more complex operators from simpler ones. While the structural operators are not original by themselves (see [10, 6] for example) up to now no study of their properties, in particular substitutability, has been conducted. We argue that the substitutability relation is important for both component providers and component deployers. It allows a component provider to build components that can be safely replaced by previous versions of the same component. On the other hand it allows component deployers to replace a component by another without worrying about compatibility issues. One of the main contributions of this article is to analyse the relation between the operand protocols and the resulting protocol while trying to focus on the *substitutability* property previously studied in [10].

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:

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

3.1 Protocol Composition

We formally define the protocol operators using the common definition of finite-state systems as 5-tuples $p = (Q, \Sigma, \delta, i^p, F)$ [11]. 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 except itself can be reached. Similar restrictions are used in approaches related to explicit protocols¹.

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 an operator for merging protocols, the union at state operator, which preserves the substitutability relation between the resulting protocol and its first operand protocol. In a second step, we define two more restricted operators — the union at the initial state and the append operators — and discuss property preservation for these operators. Then, we present an insertion operator which does not preserve any property in general but allows arbitrary protocols to be constructed as a protocol composition of the first four protocols operators. Finally, we present a first step towards the description of a protocol operator for the abstract state of the protocol.

The correctness property we are considering in this paper is protocol substitutability, which is defined when one protocol can be substituted for another one. We adopt Nierstrasz' notion of request substitutability [10] ensuring that if p is substitutable for q then

¹ Nierstrasz [10], 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.

p cannot 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).

Protocol operators are subject to a recurring problem with regard to substitutability: operators may introduce non-determinism which violates 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.

Union at state. We start defining a general protocol that allows us to insert a protocol into another at a specific state. The resulting protocol will have the same structure than the first operand and from the target state it will be able to accept sequences that are accepted from the initial state of the second operand protocol. Despite its generality, this protocol has the particularity of producing a result that is substitutable for its first operand. This operator is useful, for example, to merge different protocols, which are each associated to a different service into one protocol at different states of a main protocol.

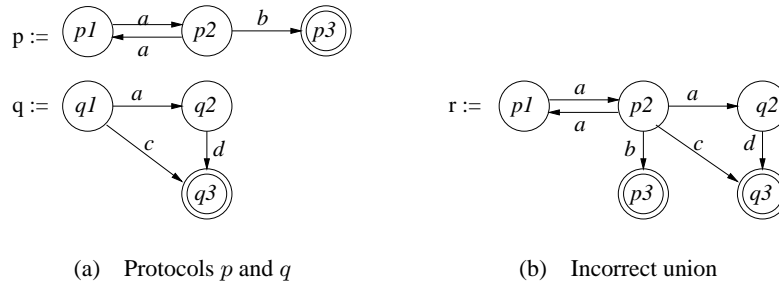


Fig. 6. Incorrect unification of q into p at state p_2

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 at the initial state of protocol p is shown in Figure 6b. After accepting the service request $a.a$, protocol p is in state p_1 and the protocol shown in Figure 6b may be in either state p_1 or q_2 , and in q_2 it may refuse a service request a , while protocol p from state p_1 cannot. The result protocol may fail to execute the sequence $a.a.a$, while protocol p cannot fail 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 from the target state (p_2 in the previous example).

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 to start from the new state. In this way different failures of states reachable by common traces are eliminated.

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

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

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

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

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

$$x \uparrow^s y \stackrel{def}{\iff} (x = y) \vee (x \uparrow^s y) \vee \exists \langle a_1, \dots, a_n \rangle \subseteq (Q^p \cup Q^q)^n : x \uparrow^s a_1 \dots a_n \uparrow^s y \quad (1)$$

The relation \uparrow^s is an equivalence relation (i.e. reflexive, symmetric and transitive; the proof can be found in [12]). We denote its equivalence classes by $[x]^{\uparrow^s}$.

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

Definition 1 (Union at state, \odot). Let p and q be two protocols, the union of protocol q at state s of protocol p , written $p \odot^s q$, is defined as the protocol $r = (Q^r, \Sigma^r, \delta^r, [i^p]^{\uparrow^s}, F^r)$:

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

$$\Sigma^r = \Sigma^p \cup \Sigma^q$$

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

This definition combines two protocols into one by using states merged by means of the relation \uparrow^s . This operator definition represents an important contribution of this paper because it ensures by construction that the resulting protocol of the union at state s can safely substitute its left operand protocol. The proof that the resulting protocol of a union operation can substitute any of its operands mainly relies on the fact that the resulting protocol inherits all the traces from its operands so it can accept all the sequences that its operands may accept. Moreover, the states of the resulting protocol are created in such a way that cannot fail to serve a request after a given sequence s of requests if the first operand is able to serve the request after s . (The formal proof can be found in [12]).

Applied to the two example protocols shown in Figure 6a, the union operator yields the result shown in Figure 7. This time, the resulting protocol will never refuse the sequence *a.a.a.*

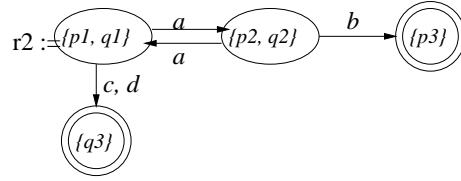


Fig. 7. Correct union of q into p at state p_2 .

Union. A particular case of the *union at state* operator allows the unification of two protocols at their initial state to result in a protocol that accepts any sequence accepted by its parent protocols. This particular union is useful, for example, to merge different per-service protocols into one protocol and has the property that the resulting protocol is substitutable for both of its operands protocols.

Definition 2 (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 :

$$r = p \circledast^{i^p} q$$

The particularity of this operator is that it ensures that the resulting protocol can be substituted for any of its operand protocols (the proof can be found in [12]). Applied to the two example protocols shown in Figure 6a, the union operator yields the result shown in Figure 8a.

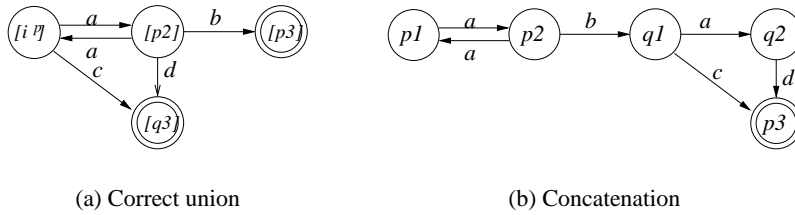


Fig. 8. Applying different operators to p and q

Concatenation. The concatenation of two protocols consists in appending one protocol to another. The resulting protocol of 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. We define this operator as a restricted variant of the union at state operator in order to get the benefit of its substitutability property.

Definition 3 (Concatenation operator \mapsto). Let p and q be two protocols and $F^p = \{f_1, f_2, \dots, f_n\}$ the set of p 's final states. The concatenation of protocol q to protocol

p , written $p \mapsto q$, is defined as follows:

$$p \mapsto q := ((p \otimes^{f_1} q) \otimes^{f_2} q) \cdots \otimes^{f_n} q$$

Figure 8b 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. In other words, $p \mapsto q$ is protocol substitutable for protocol p but not for protocol q .

Insertion. Let us consider a very general protocol operator which allows the 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 9a. 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 9b.

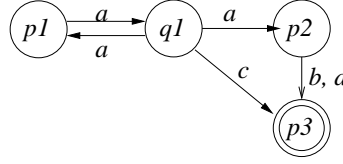


Fig. 9. Inserting q at p : $p \odot_{\{(q_2, p_2), (q_3, p_3)\}}^{p_2} q$

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.

This operator is not very natural in terms of protocol programming, but its main purpose is to allow us to express any protocol as the protocol composition of the four first protocols presented up to now. Let us define the insertion protocol operator more formally:

Definition 4 (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:

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

This definition constructs an automaton preserving the states of both automata except for unified states, i.e., states in $Dom(I)$. 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.

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 an identity constraints. This operator can be defined as follows:

Definition 5 (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 [13] is a white-box component model based on Java [14]. 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 [15], 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 to 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 encapsulate information about a state change of an *event source* bean. The event source bean must implement two methods for adding and removing listeners.

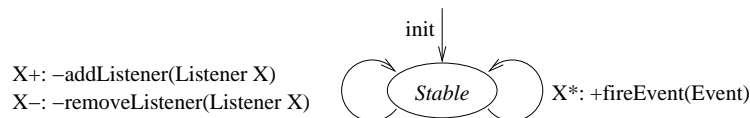


Fig. 10. Protocol for basic event management

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 10. The protocol is defined by one state with three transitions constrained by identity constraints. The transition labelled 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 labelled `removeListener(...)` allows a component to request its removal from the list of listeners provided that it has been previously registered. Finally, the transition labelled `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 11. Basically, bound properties reuse the protocol for basic event management and add one state which is

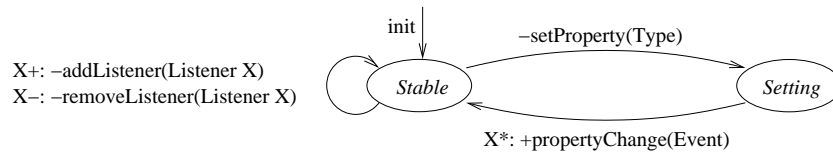


Fig. 11. Protocol of bound properties

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.

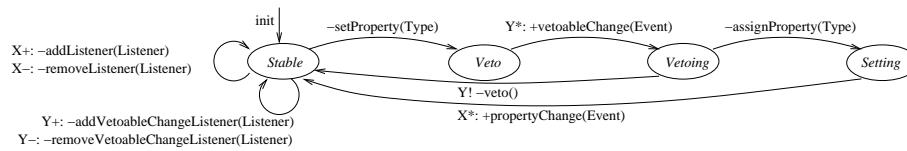


Fig. 12. Component protocol of a constrained property.

The protocol of constrained properties can be made explicit as shown in Figure 12. 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* 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

```

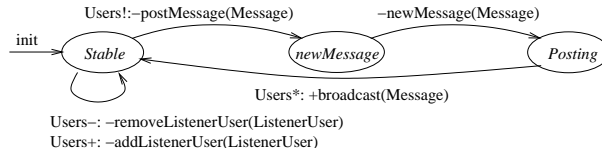
protocol Veto {
  Veto    :: Y*      : +vetoableChange(Event) --> Vetoing
  Vetoing :: +assignProperty(Type)           --> Approved
          Y!      : -veto()                  --> (init) Start
}
  
```

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*.

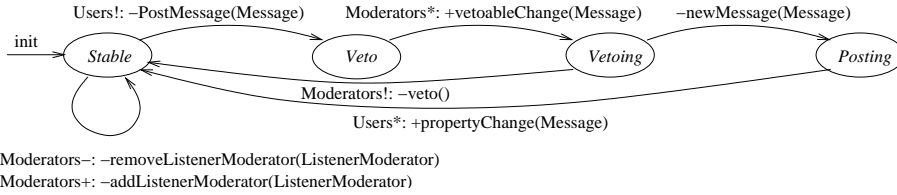
The constrained protocol can then be written as a protocol composition between the bound protocol and the *Veto* protocol by inserting the *Veto* protocol after the state *Stable* and redirecting transitions as follows:

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

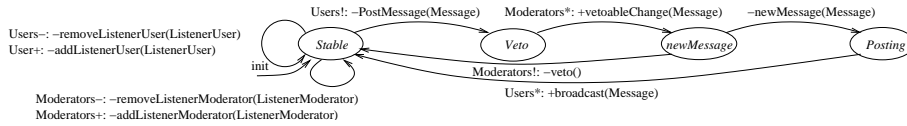
We can formally prove the correctness of this construction by equational reasoning; the proof is given in [12].



(a) The posting protocol.



(b) The moderator protocol.



(c) The resulting protocol.

Fig. 13. Moderators protocol.

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 broad-

casting of an inadequate message. This can be implemented by constraining the messages to be sent with the vetos of moderators. Figure 13 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:

$$ModeratedPosting = Posting \odot_i^{Stable} Moderator$$

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

$$Chat = ModeratedPosting \oplus SearchMessage \quad (2)$$

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

Fig. 14. Login and Logout protocols.

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 14 as follows:

$$SecureChat = Login \mapsto^X Chat \quad (3)$$

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

5 Extending the EJB security model

Enterprise JavaBeans (EJB) [1] is an industrial-strength component model, which supports the development, deployment, and management of transactional business systems using distributed components implemented in Java. In this section we discuss several limitations of the security model of EJB and we show how can we use explicit protocols to improve expression of access-control policies at the interface level.

5.1 The EJB security model and its limitations

The EJB security model is a role-based access-control model that permits deployers to configure and parameterize security in a *black box* fashion. An EJB server can authenticate a client (using an authentication service, such as the JAAS service, for example) and then authorize the client to call certain of its services. The authorization process is based on a mechanism that maps a set of identities to a role with associated authorizations that are declaratively specified in the *deployment descriptor* file. At runtime a bean is subject to security checks performed on the server side by the bean container, which has been configured based on the deployment descriptor.

During bean development, the bean provider can implement security-related code directly within the bean, effectively working in a *white box* fashion. To this end, the EJB component model provides some methods in the `EJBContext` class to query the container about some security-related information, such as the identity of the caller of a method (`getCallerPrincipal()`) or to check whether a caller belongs or not to a given role (`isCallerInRole(String)`). One of the main goals of EJB's security model is to provide a mechanism for specifying policies at deployment time and to avoid the insertion of security-related code into the bean after the bean development phase. This way, providers are encouraged to develop beans without mixing security policies with the business logic and the application of security policies is left to the deployer of an application.

The EJB security model is well suited for the expression of security policies to control method calls according to the caller's identity. This is appropriate for implementing many security policies for distributed resources. However, we argue that more complex policies break down the overall goal of separating the security logic from the bean code. Explicit protocols enable three natural extensions of the EJB security model that allows more flexible security policies to be defined while respecting the goal of separation mentioned above:

1. Explicit protocols enable access to be controlled on the basis of information about sequences of methods. For example, in EJB it is possible to authorize identities belonging to a role X to execute methods m_1 and m_2 , but it is not possible to restrict access of an identity to call m_2 only if it has already called method m_1 (which can be expressed using explicit protocols as $X! : -m_1 - m_2$). This limitation does not allow programmers to specify a policy such as *a client must first get the value to incrementing it later*.
2. EJB does not provide support for specifying the interaction between roles when calling a sequence of services. Suppose, for example, that a role X can call methods m_1 and m_2 and role Y can call method n . There is no explicit support for stating policies such that role X calls method m_1 , then role Y calls n and, finally, X calls method m_2 (using explicit protocols: $X! : -m_1$; $Y! : -n$; $X! : -m_2$).
3. Roles are static in the sense that it is not possible to add or to remove dynamically identities from roles. This is because roles and identities are associated at deployment time.

This three extensions directly put at work the definition of identity constrain terms presented in section 2.1. In order to illustrate how explicit protocols can be used to overcome these limitations while preserving a black-box model of access control, let us have

another look at the example of the distributed chat server. Consider the collaboration of two different entities belonging to different roles: clients and moderators, where a client may create a forum and become moderator of that forum. Let the forum creation be supported by the chat server through the following services: `askForForum(Forum)`, which allows a client to propose the creation of a forum; `verifyRequest()`, which is used to validate requests with respect to some criteria; and `openForum(Forum)`, which allows the instantiation of a forum. We assume that clients have two methods: `accepted(Forum)` and `refused(Forum)`, which are invoked to inform a client if a forum-related request has been accepted or refused. We would like to state the following *forum-related policies* over that set of services:

1. Limit the invocation of `askForForum(Forum)` and `openForum(Forum)` to *clients*, and access to `askForForum(Forum)` to clients that have already called `openForum(Forum)`.
2. Limit calls to `verifyRequest()` to moderators and additionally ensure that `verifyRequest()` is executed after `askForForum(Forum)` is performed.
3. Require that `accepted(Forum)` and `refused(Forum)` are called only on clients that have requested the creation of a forum.

The EJB security model is not appropriate to implement these policies. For the first and second policies, it is possible to restrict the call of the methods to authenticated entities that belong to the corresponding roles (Client and Moderator). This can be done in a declarative way by specifying the identities belonging to each role in the deployment descriptor. However, it is not possible to specify in the deployment descriptor constraints over sequential executions of methods. To implement the second and third policies, in EJB, it is necessary to make an analysis of previous methods calls with respect to the current method by introducing code for managing sequences of methods calls.

5.2 Using protocols to enhance the EJB security model

Figure 15 shows a protocol implementing the three forum-related policies, governing the creation of new forums. The server restricts the execution of the method `askForForum(Forum)` to the same client c that afterwards requests `openForum(Forum)`. The protocol limits requests `verifyRequest()` to moderators (represented by the identity term M) and expresses that a client must have requested the creation of a forum before. Only after the request `verifyRequest()`, the server is allowed to send an answer to the client who has requested the forum's creation. Finally, only after the reception of an acceptance notification from the server, the client can request the initialization of the forum by the server.

Once again we can show how to extend the chat server protocol by unifying the secured version of the chat server, defined in Equation 3, at the state *Stable*, with the new service protocol:

$$NewChat = SecureChat \circ^{Stable} ForumCreation$$

By construction, substitutability w.r.t. the secured chat server protocol holds, and we can thus conclude that existing clients are still compatible with this new version of the server.

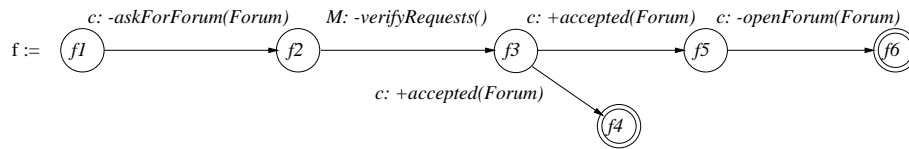


Fig. 15. Protocol of a Forum creation request

We have given evidence that component with explicit protocols can provide a better solution for the implementation of sophisticated policies. First, protocols are expressed at the interface level and security policies can be expressed separately from code of the client and the server beans and their expression can be done in the deployment descriptor, thus not breaking EJB security model. Second, protocols allow us to describe sequencing constraints between methods and roles. Third, protocols allow a more flexible treatment of identities and roles, such as adding and removing identities from roles dynamically. Finally, the construction operators and associated substitutability property are also useful within a black-box component model: unnecessary generation of adaptors and modification of the client/server code is avoided.

6 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 [10] to investigate service availability of objects using CSP [16] 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 [17] 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 [18] 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 [9]. However, they do not consider property preservation of such operators. Yellin and Strom [6] 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 [7] 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 [3], for instance, has been applied to the specification of components. Similarly, message sequence charts [19] (for an application to components see the PhD thesis of Wydaeghe [8]) is a trace language which can be used to describe interactions among components based on finite-state systems. Architectural description languages, such as Wright [20], 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.

7 Conclusion

Extending previous work on finite-state based protocols for objects and components, we proposed three 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 to restrict protocol transitions based on the identity of collaborating components. Finally, we validated the use of the proposed techniques by applying them to Sun’s component models, JavaBeans and Enterprise JavaBeans. We were 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. As to EJB, we have shown that explicit protocols at the interface level allow the declarative formulation of more flexible access-control policies while preserving EJB’s black-box properties.

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 to make protocol construction more flexible. Furthermore, our investigation of separated state information only constitutes a first step of this concept and should be pursued. Finally, we plan to test our ideas with real-life EJB.

References

- [1] DeMichiel, L., Yalçinalp, L., Krishnan, S.: Enterprise JavaBeansTM Specification. SUN Microsystems. (2001) Version 2.0, Final Release.
- [2] Kruchten, P.: Rational Unified Process: an Introduction. Addison-Wesley, Reading, Massachusetts, USA (1998)
- [3] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. 1 edn. Addison-Wesley, Reading, Massachusetts, USA (1999)
- [4] Harel, D.: Statecharts: A visual formalism for complex system. *Science of Computer Programming* **8** (1987) 231–274
- [5] Mallet, F., 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., Milan, Italie (1999) 214–222
- [6] Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. *ACM Transactions of Programming Languages and Systems* **19** (1997) 292 – 333
- [7] de Alfaro, L., Henzinger, T.A.: 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 (2001)
- [8] Wydaeghe, B.: PACOSUITE, Component Composition Based on Composition Patterns and Usage Scenarios. PhD thesis, Vrije Universiteit Brussels (2001)
- [9] Plasil, F., Visnovsky, S.: Behavior protocols for software components. In: *Transactions on Software Engineering*, IEEE (2002)
- [10] Nierstrasz, O.: Regular types for active objects. In Nierstrasz, O., Tsichritzis, D., eds.: *Object-Oriented Software Composition*. Prentice Hall (1995) 99–121
- [11] Hopcroft, J.E., Motwani, R., Ullman, J.D.: *Introduction to Automata Theory, Languages, and Computation*. Second edn. Addison Wesley (2001)
- [12] Farias, A., Südolt, M.: A component model with explicit protocols. Technical Report 02/4/INFO, Ecole des Mines de Nantes (2002)
- [13] Hamilton, G.: Java beans API specification. Technical report, Sun Microsystems (1997)
- [14] Sun Microsystems: The Java language: An overview. Technical report, Sun Microsystems (1995)
- [15] Eugster, P., Guerraoui, R., Damm, C.: On objects and events. In: *Proceedings OOPSLA*. (2001)
- [16] Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. *Journal of the ACM* **31** (1984) 560–599
- [17] Van Den Bos, J., Laffra, C.: PROCOL: a parallel object language with protocols. *ACM SIGPLAN Notices* **24** (1989) 95–102
- [18] : SOFA project. (<http://nenya.ms.mff.cuni.cz/thegroup/SOFA/sofa.html>)
- [19] Rudolph, E., Graubmann, P., Grabowski, J.: Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems* **28** (1996) 1629–1641
- [20] Allen, R.J.: A Formal Approach to Software Architecture. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh (1997)