A Formal Component Model with Explicit Symbolic Protocols and its Implementation in Java

Sebastian Pavel[1], Jacques Noyé[1], Pascal Poizat[2] and Jean-Claude Royer[1]

[1] OBASCO Project-team, École des Mines de Nantes-INRIA,
4 rue Kastler, 44307 Nantes cedex 3, France
{spavel, noye, jroyer}@emn.fr
[2] LaMI, UMR 8042 CNRS-Université d'Évry Val d'Essonne, Genopole
Tour Évry 2, 523 place des terrasses de l'Agora, 91000 Évry, France
poizat@lami.univ-evry.fr

Abstract. Component-Based Software Engineering (CBSE) has now emerged as a discipline for system development. Important issues in CBSE such as composition incompatibility detection and (dynamic) adaptation can only be addressed with the help of formal component models with Behavioural Interface Description Languages (BIDLs) and explicit protocols. The issue is then to fill the gap between such high-level models and implementation. This paper suggests to do so by relying on Symbolic Transition Systems (STSs). It describes a component model with explicit symbolic protocols based on STSs, and its implementation in Java. This implementation is based on controllers that encapsulate protocols and channels devoted to (possibly remote) communications between components.
keywords: CBSE, Behavioural IDL, Explicit Protocols, Symbolic Transition Systems, Java, Controllers, Channels

1 Introduction

With the increase in complexity of software systems in the last few years, the Component-Based Software Engineering (CBSE) approach has emerged as a discipline that yields promising results such as trusted and/or Off-The-Shelf components (COTS), improved component reusability, semi- or automatic composition and adaptation of components into architectures, expressive middleware, and so on. A major drawback of the mainstream industrial approach to CBSE is that it has mainly focused on programming/low-level features making it difficult to reason on problematic issues hidden behind the CBSE promising results.

Recent meta-descriptive approaches such as the OMG-Model Driven Architecture and Aspect-Oriented Software Development promote the return to abstract models and clear separation between the functional (or business) and non functional (or technical, *e.g.* real-time constraints or synchronizing policies) aspects of a system. Orthogonally, Architectural Description Languages [28] and

Coordination Languages [31] are also increasingly promoted in the CBSE community. Both are interesting with regard to their ability to abstract the systems and provide better/simpler reasoning mechanisms, but also because this activity can be formally grounded and then equipped with tools.

The first important issue when designing a component model is related to the definition of the component *interfaces* using Interface Description Languages (IDLs). This has been initially addressed by industrial component infrastructures to statically (*i.e.* at compile time) generate skeletons and stubs for distributed components. More recently, this issue has turned to be at the core of the most challenging issues in CBSE: component validation and trusted components, (dynamic) adaptation, negotiation and choreography. The limit of IDLs based on signature types has now been demonstrated. For instance, type correct communicating components may deadlock because they do not have compatible protocols. It is therefore now widely accepted that IDLs have to take into account *behavioural protocols*, yielding Behavioural IDLs (BIDLs). These protocols may be used either as a piece of documentation for the components, in a design-by-contract process, to compose and check connections between components or even build adapters [12, 41] when component interfaces do not match. Several formal models dealing with such IDLs and protocols have been proposed. However, as components exchange data using their provided and required services, formal models have to take data into account while managing potential state-explosion problems.

In this paper, we suggest using Symbolic Transition Systems [10, 14] (STSs) as the basis of a BIDL. This expressive formalism makes it possible to control state explosion thanks to the use of guards and complex typed parameters associated to the transitions. We show how STSs can be used as explicit protocols in a hierarchical component model supporting multiple interfaces with heterogeneous services (synchronous and asynchronous communications). An important property of this model is that the protocols are verified by construction. We present a Java implementation of the model whereby the code of a primitive component is synthesized from an STS protocol and Java code. A controller intercepts the communications and calls the related service of the inner Java code according to the protocol. The subcomponents of a compound component communicate via channels dealing with the guarded synchronous and asynchronous communications.

The paper is organized as follows. Sect. 2 describes our component model. Sect. 3 introduces a simple case study that will illustrate our presentation. Then Sect. 4 explains how our model can be implemented in Java to deal with explicit protocols and explicit component binding mechanisms. Sect. 5 presents related approaches and Sect. 6 concludes.

2 STS-oriented Component Model

Like any component model, our model builds on the ADL ontology [28]: architectures or configurations made of components (with ports) and connectors (with roles), and bindings between component ports and connector roles.

The specificities of our model are: heterogeneous interfaces incorporating typed services of different kinds, explicit behavioural protocols, and the use of Symbolic Transition Systems. We rely on a simple binding mechanism rather than on complex connectors. For the time being, we only consider one-to-one, one-way messages. We are also dealing only with static architectures.

2.1 Primitive Components

Primitive components are made up of a type name, named interfaces, a behavioural protocol and an underlying data type, or implementation, on which the component relies to achieve the services it provides (each service corresponds to a function or method in the implementation).

Interfaces. Components interact through named interfaces, each one corresponding to a communication port. An interface corresponds to a set of services A service is given as a name (unique within a given interface) and a type which corresponds to the type of the values received/emitted by the service. A service can be provided or required, and for each of them, communication can be either synchronous or asynchronous. We consider reliable message sending based on synchronous call, asynchronous messages are realized through the use of mailboxes and decoupling between message receipt and execution. This approach is based on [38], which gives a more precise discussion about the need for asynchronous communications. One specificity of our approach is that the operations related to mailboxes are transparent at this level of abstraction.

The graphical notation for interfaces and services is given in Fig. 1. A textual notation is also proposed in the form of a component/protocol description language. The grammar constructing this language is presented in Appendix A. In the graphical notation, boxes correspond to synchronous services and circles to asynchronous ones. Black symbols denote required services (emissions) and white ones denote provided services (receipts). Our syntax for interfaces was inspired by various component graphical notations, mainly from the Olan ADL [6] (for the port symbols) and from process algebras such as LOTOS [39] (for the input/output event schemes). The textual notation of interfaces is given, with respect to the Java language, in Sect. 4. The keywords corresponding to service kinds are: `require` for required synchronous services, `provide` for provided synchronous services, `notify` for required asynchronous services and `react` for provided asynchronous services.

Protocols. A known drawback of several component models is their lack of component behavioural protocols. It is well known that purely static (signature based) interfaces are not sufficient to detect inconsistencies in two interacting

component protocols. Moreover, behavioural pieces of information on components are also needed to perform tasks such are dynamic adaptation [12] or negotiation. Such protocols can be expressed using different formalisms: automata [41], behavioural types [21], process algebras [7], temporal logics [16]. We chose the Symbolic Transition Systems framework.

Symbolic Transition Systems (STS) [10, 22] have initially been developed as a solution to the state and transition explosion problem in value-passing process algebras using substitutions associated to states and symbolic values in transition labels. Our STSs (see example Fig. 4) are a generalisation of these, associating a symbolic state and transition system with a data type description [5, 14, 36]. This description may be given using algebraic specifications [14, 36], model-oriented specifications [5] or even Java classes. STSs can be related to statecharts (see [33] for details) but are simpler as far as semantics is concerned.

In the context of this formal presentation, we will take into consideration abstract descriptions given using algebraic specifications. [15, 36] explain how java classes can be derived from them. We will give here only the necessary insight into these specifications, see [4] for more details. A *signature* (or static interface) Σ is a couple $(\mathcal{S}, F)$ where $\mathcal{S}$ is a set of *sorts* (type names) and F a set of function names equipped with *profiles* over these sorts. If R is a sort, then Σ_{R} denotes the subset of functions from Σ with result sort being R. X is used to denote the set of variables. From a signature Σ and from X, one may obtain *terms* , denoted by $T_{\Sigma,\mathrm{X}}$. The set of *closed terms* (also called ground terms) is the subset of $T_{\Sigma,\mathrm{X}}$ without variables, denoted by T_Σ. As usual, $\mathcal{P}()$ is used to denote the powerset of a set, and $\vec{x}$ denotes a vector. An *algebraic specification* is a pair (Σ, Ax) where Ax is a set of axioms between terms of $T_{\Sigma,\mathrm{X}}$. Models of specifications are Σ-algebras. They relate each sort s in Σ to a set of values denoted by $s^\sharp$ (interpretation of s), and each function name f in Σ to a function over these values denoted by $f^\sharp$ (interpretation of f). We are only interested in *reachable* algebras, that is those where each value can be obtained from (at least) a ground term in T_Σ. It is assumed that the data type provides a computable equality function to check whether two ground terms represent the same value.

We may then give a structural definition of our STSs, see Sect. 3 for textual syntax and graphical representation.

Definition 1 (STS). *A Symbolic Transition System (STS) is a tuple* $< D,\ (\Sigma,\ Ax),\ S,\ s_0,\ v_0,\ T >$ *where:*

- (Σ, Ax) *is an algebraic specification,*
- D *is a sort called* sort of interest *defined*[3] *in* (Σ, Ax)*,*
- $S = \{s_i,\ 0 \leq i \leq N\}$ *is a finite set of states,*
- $s_0 \in S$ *is the initial state,*
- $v_0 \in D$ *is the initial value,*
- $T \subseteq S \times \Sigma_{Boolean} \times \Sigma_{\mathrm{D}} \times \mathcal{P}(X) \times S$ *is a finite set of transitions,*
- Σ *contains a set* $\{P_{s_i}\}$ *of state predicates which are functions which associate a state in* S *to each value in* D*,*

[3] More exactly, (Σ, Ax) is an *algebraic presentation* for sort D [4].

– $\forall v, v' \in D^\sharp,\ v \approx_{D^\sharp} v' \widehat{=} (\exists! s_i \in S,\ v \in s_i \wedge v' \in s_i)$, *is an equivalence relation.*

STS transitions are tuples $(s,\ \mu,\ \lambda,\ \vec{x},\ t)$ for which s is called the source state, t the target state, μ the guard, λ the label and $\vec{x}$ the variables. LTS labels are closed terms. Our STS consider labels as operation calls. Moreover, there are guards in transitions which denote boolean functions. Hence a STS is a LTS if and only if the λ in transitions are functions from D to D, all the $\vec{x}$ are empty, and all the μ guards are *true*.

The semantics of STS is given in terms of configurations and transitions between configurations.

Definition 2 (Configuration). *A* configuration *of an STS* $< D,\ (\Sigma,\ Ax),\ S,\ s_0,\ v_0,\ T >$ *is a pair* (s, v) *with* $s \in S$ *(control state), and* $v \in D^\sharp$ *(value).*

Equivalence between two configurations is defined as equality on control states and equality on values (this is why a computable equality function is needed in the algebraic specification part of the STS). The set of all possible configurations of an STS is denoted $G(STS)$. Let $\vec{x}$ be a vector of variables, $\sigma_{\vec{x}}$ denotes a substitution over this vector, $x\sigma$ with $x \in \vec{x}$ denotes the value of x in this substitution and $\vec{x}\ \sigma$ denotes the application of σ to the whole vector (that is a vector of values).

Definition 3 (Configuration Transition). *A (fireable) configuration transition for a configuration* $(s,\ v)$ *of a STS* $< D,\ (\Sigma,\ Ax),\ S,\ s_0,\ v_0,\ T >$ *is any tuple*
$((s,\ v), \mu,\ \lambda,\ \sigma_{\vec{x}},\ (t,\ v'))$ *such that* $(s,\ \mu,\ \lambda,\ \vec{x},\ t) \in T$, $\mu^\sharp(v, \vec{x}\ \sigma) = true_{\text{Boolean}}$, *and* $v' = \lambda^\sharp(v, \vec{x}\ \sigma)$.

The usual notions over LTS (*e.g.* traces, reachability tree and reachability graph) are naturally extended to STS using configuration transitions and taking $(s_0, v_0^\sharp)$ as initial configuration. The semantics associated to an STS is a configuration relation which is the set of the configuration transitions of the STS.

Implementation. The implementation part of components is an encapsulated datatype element of some kind on which the behavioural protocol relies to achieve operations corresponding to its transitions. An assumption is that this datatype does not only implement the interface services but also that it does it in a way that is compliant with the protocol. In a purely formal specification approach [14, 33], the implementation is given as an algebraic datatype. In such a case, derivation mechanisms originating from the Graphical Abstract data Type (GAT) formalism [15, 36] makes it possible to obtain semi-automatically a compliant datatype. Here, the implementation part is given as a Java application (see Sect. 4). Translation from GAT datatypes to Java code could be used [15]. Code analysis could also be performed to check protocol compliance. This is out of the scope of this paper.

Syntax. The graphical syntax of a component is given in Fig. 1. The textual syntax depends on the target language. The instantiation for the Java language is given in Sect. 4.

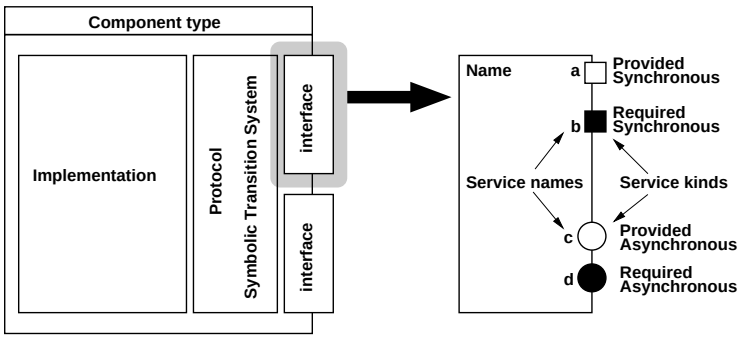

Fig. 1. Components Syntax

Semantics. Various operational and denotational semantics have been defined for variants of STSs, see [1, 14, 27]. The semantics of a component corresponds to the semantics of its STS which may be expressed as a configuration graph (see [27]). A configuration graph is a LTS with values associated to states, thus the usual notions (*e.g.* traces, reachability tree and reachability graph) are naturally extended to an STS using configuration transitions. This formal framework enables component-based analysis [12].

2.2 Compound Components

Compound components are component assemblies given as a set of identified and typed subcomponents and a set of bindings between these component interfaces. Communication is binary (one sender, one receiver) and oriented (no rendezvous as in process algebras [7, 39]). The matching between services of two bound interfaces is done through name matching (this can be achieved using renaming), that is, given two bound interfaces I_1 and I_2, each required service s of I_1 (resp. of I_2) corresponds to a provided service s (same name) in I_2 (resp. I_1). Moreover, service types and kinds (asynchronous/synchronous) must correspond too. In order to have compound components, an interface is given for the compound and bindings (exports) are defined between this interface and the subcomponents interfaces. Any interface of the subcomponents in a compound component that is not bound can be exported.

Syntax. A compound component may be given either in its graphical form (Fig. 2) or its textual form (see Sect. 4 for its Java version).

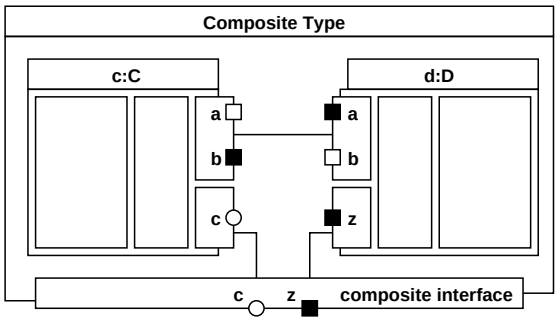

Fig. 2. Compound Syntax

Semantics. Formally, the semantics of composites is obtained by adapting the synchronous product to STS [1, 14, 38].

2.3 Model Analysis

Models expressed using our formal model may be analyzed. Such a process may either rely on automatic checking tools or on manual (tool-assisted) theorem proving.

Automated Techniques. Automated techniques are either based on model checking [16], behavioural equivalence [7] or animating tools which have already been proved efficient in the protocol specification community. Model checking tools check properties expressed in a modal logic against the system behaviour. Such properties can be either *safety* (something wrong never happens, *e.g.* no deadlock) or *liveness* (something good may happen) properties. Recent extensions in behavioural languages led to corresponding modal logics and model checkers, taking into account time, probabilities or spatial information. Behavioural equivalence tools can be used to check protocol consistency, bindings correctness or component refinement. The use of such formal languages, techniques and tools for component-based analysis is more detailed in [34].

However, all these techniques rely on very abstract forms of behaviours.In the context of value passing/exchange and datatype encapsulation, the tools behavioural languages loss of abstraction cause state explosion problems. This issue led to the definition of the STS framework, but checking techniques and tools have to be adapted to take such abstract definitions into account.

Recent works adapted bisimulation equivalence to STS [10] fully taking into account their abstract form.

As far as model checking is concerned, over-approximation techniques, such as evaluating all guard to `true` and bounding datatypes is a simple solution to get an finite transition system which may be analyzed using usual techniques.

We are currently working on abstraction techniques for STS in the context of mailbox analysis for asynchronously communicating components. This

frameworks aims at taking into account different mailbox policies or even shared dataspaces used in coordination languages [31]. Mailbox analysis aims at checking if mailboxes are bounded, and if so by which size. These are an interesting family of properties as mailbox unboundedness may cause availability loss. Moreover, whenever a component mailbox has been proven bounded by an integer `k`, this information may be used to simplify the component protocol (it is well known that unbounded mailboxes are a cause of state explosion). Results have been obtained in [27, 30] where different mailbox policies (the usual FIFO one and DICO, an abstraction of it) and the corresponding component configuration graphs have been studied. Among different results, we have shown in [27] that FIFO-boundedness is not decidable, that DICO-boundedness is decidable, and we have given a necessary condition for mailbox boundedness of component architectures in the context of possibly heterogeneous mailboxes. Such results could be used conjointly with model checking or to simplify the component protocols specification and the proof in a given architecture before using other analysis techniques [30].

A perspective could be to adapt abstract interpretation techniques used in the security protocol framework [9] to component-based verification. However, this would require first to gather all interesting properties and then build inference systems (in the logical programming sense) corresponding to the STS transitions for each family of properties. Techniques developed in the test community [8] could also be used to build abstractions of the system from a given logic property. Moreover, we are currently working the the adaptation of [8] testing techniques to compositional testing in the framework of a national research project. Our idea is to use the system structure to build simpler abstractions for its subcomponents, test properties on these abstractions and lift the results up at the system level.

Theorem Proving. We are also interested in the use of theorem provers to check STS properties. We especially focus on the development of automatic proof strategies (depending on the properties form) and interaction between theorem proving and model checking. In [35, 30] we have shown that the PVS theorem prover could be used to conduct proofs on our models. To achieve this, component STS and synchronizing information are used to build, adapting the synchronous product technique, a global STS which is then encoded as an algebraic datatype on which proofs are undertaken using PVS adapted strategies. This work has then been extended in [37] where we build a temporal logic dedicated to STS, CTL*Data, and develop techniques to prove them using PVS. Deadlock freedom, but also properties involving datatypes have been proved in this context.

3 The Flight Reservation System Example

In order to demonstrate our proposal, we present a simplified flight ticket reservation system. The reservation system (see Fig. 3) contains a `Company` component, a `Bank` component, and a `Counter` component. The `Company` is responsible for

proposing the available flights corresponding to a particular request. The `Bank` manages the bank accounts of the clients. The `Counter` is the most important component as it receives the requests from clients and then coordinates the search and confirmation (by interacting with the `Company`) and the payment (by calling the `Bank` services) of the confirmed flight. As a coordinator, the `Counter` exposes three interfaces: `bookingIntf`, `paymentIntf` and `orderingIntf`. While the first two interfaces are used to connect the `Counter` to the `Company` and the Bank, respectively, `orderingIntf` is used to interact with the clients. The types associated to these interfaces are given in Fig. 5.

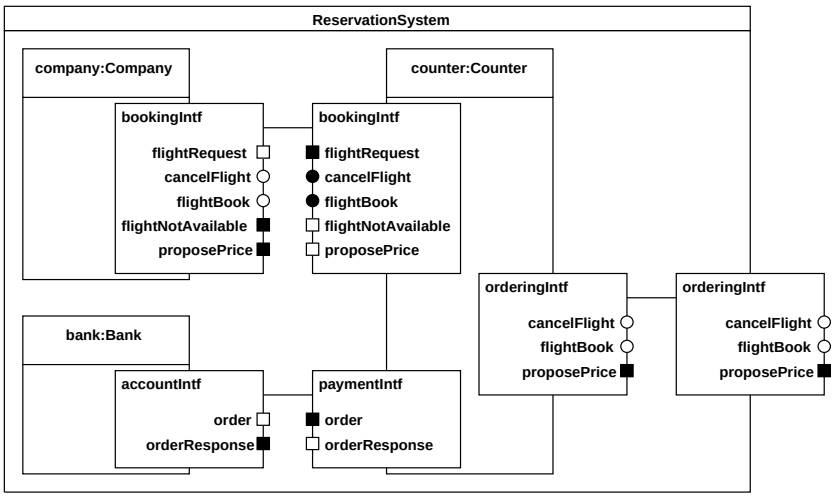

Fig. 3. A Simplified Ticket Reservation System

Once connected, the `Company`, the `Bank`, and the `Counter` form a compound component called `ReservationSystem`. A client component does not have to know the internals of this component. It will only communicate through the interfaces exposed by `ReservationSystem`. The interaction between a client and the system is actually implemented within the `Counter` component, the exposed interface is `orderingIntf`. Once the architecture has been built, all requests coming from a client are transferred to the `orderingIntf` in the `Counter`.

In order to facilitate the understanding of STS protocols associated to components we propose a graphical description. The protocol described in Fig. 4 represents the allowed behaviour associated to the `Company` component. The protocol consists of three states and several transitions between these states corresponding to the messages that are received and emitted to and from the component. In addition, the protocol specifies the messages ordering. For example, a booking (`flightBook`) or cancellation (`cancelFlight`) message cannot be exchanged before a `proposePrice` message is received. The `flightBook` and `cancelFlight` messages are guarded (a boolean expression between square

brackets). They will be processed only if the corresponding guard (depending on message parameters) is true. Received messages are preceded by a *?* sign. A *!*

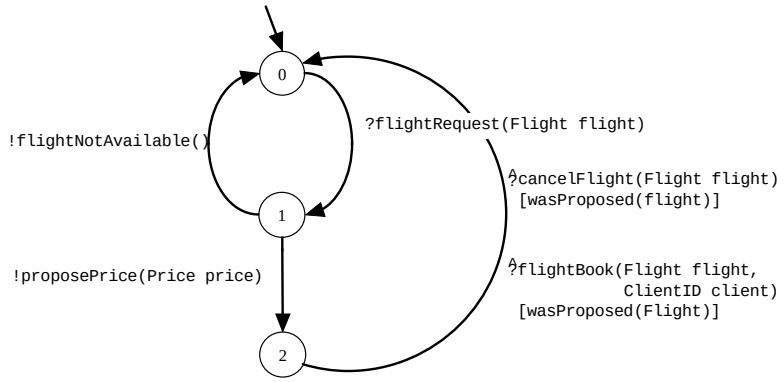

Fig. 4. The Company STS Protocol

sign precede a sent message and the $\hat{?}$ sign denotes an asynchronous message.

3.1 Textual Description

Graphical descriptions of protocols are useful for human understanding of dynamic systems. However, automated computation requires a textual representation. We have defined a minimal component language relatively close to Java to describe the interfaces and the protocol of a basic component. The grammar for this language is presented in Appendix A. Fig. 5 depicts the definition of the `Company` component using the proposed language. A definition for a basic component comprises one or more `Interface` sections used to define the services of each interface of the component. The second section defines the operations representing the guards used in transitions (`wasProposed` in Fig. 4). The last section describes the STS protocol we want to associate to this component. The `Protocol` section contains the list of states of the protocol including the initial state and the final states, if any, and the list of transitions. The protocol associated to the `Company` has no final state meaning that the execution runs forever.

The definition of compound components is slightly different. In order to obtain the `ReservationSystem` compound component from the example we would write the code presented in Fig. 6. After declaring the subcomponents, the `connect` primitives are used to put in correspondence the components interfaces. This primitive makes no assumption with regard to the type of connection (local, remote, etc) employed to actually connect the implementation components at runtime. This information is given at deployment time. The `Interface` section defines the interfaces to be exposed by the compound component. In

```
component Company {
  interface bookingIntf {
     PROVIDE void flightRequest(Flight flight);
     REQUIRE void proposePrice(Price price);
     REQUIRE void flightNotAvailable();
     REACT   void flightBook(Flight flight, ClientID client);
     REACT   void cancelFlight(Flight flight);
  };

  guards {
     boolean wasProposed(Flight flight);
  };

  protocol {
    states {0 (initial), 1, 2};
    transitions {
     0:flightRequest                    -> 1;
     1:flightNotAvailable               -> 0;
     1:proposeFlight                    -> 2;
     2:flightBook(flight, client)
       [wasProposed(flight)]            -> 0;
     2:cancelFlight(flight)
       [wasProposed(flight)]            -> 0;
    };
  };
};
```

Fig. 5. The Company Textual Description

```
compound ReservationSystem {
  Company company;
  Counter counter;
  Bank bank;
  connect company.bookingIntf, counter.bookingIntf;
  connect bank.accountIntf, counter.paymentIntf;
  Interface orderingIntf{
    export Counter.orderingIntf;
  };
 };
```

Fig. 6. The ReservationSystem Textual Description

the proposed example, the interface to be exposed is the same as that of the `Counter` subcomponent but a combination of interfaces of subcomponents into a new interface of the compound is possible.

4 Model Implementation

The general idea is depicted in Fig. 7. The purpose is to attach an STS protocol to an already defined "component" (further referred as *bare component*). After attaching the protocol, the result is another component (further referred to as *controlled component*) with the same functionality as the initial one and in addition a mechanism to check and impose the specified protocol. A bare component would connect into an architecture using simple binding mechanisms (if the correspondent is local) or RMI connections (if the correspondent is remote). A controlled component would connect using a special connection based on communication channels as we will see later in this section.

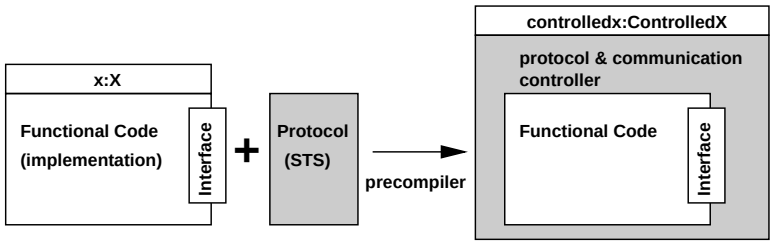

Fig. 7. General Idea

The guidelines to follow in order to implement a component in Java are discussed in Subsect. 4.1. The approach we propose in order to attach a given protocol to a given component is described in Subsect. 4.2.

4.1 Bare Components

One simple solution to implement a component into Java is to use the `package` feature provided by this language. In order to achieve the hard boundaries we need for components, we use an approach close to the one presented in [3]. This approach imposes some restrictions. First, a component must have at least one class that represents the component's interface. Second, these interface classes are the only public classes in the `package`. Third, only interface classes can have public methods. Fourth, the internal references are invisible from outside the component boundaries.

The clear advantage of such an approach is that the access to a component is possible only through its clearly specified interfaces. On the other hand, the restrictions we impose does not have a great impact on how the component is

implemented. At execution, a component instance may have one or more objects, one or more active entities, etc.

4.2 Protocol Implementation

To integrate protocols expressed as STSs, we have identified two major approaches. First, we could modify the code of a bare component in order to behave as specified by the protocol. One solution would be to use AOP [23], or bytecode altering techniques [13] to create a new Java class hierarchy implementing the component with the protocol. The second approach does not modify the initial code but rather creates a framework of classes around the initial code. Once instantiated, this framework will become a component (with the same functionality as the initial one) integrating the specified protocol.

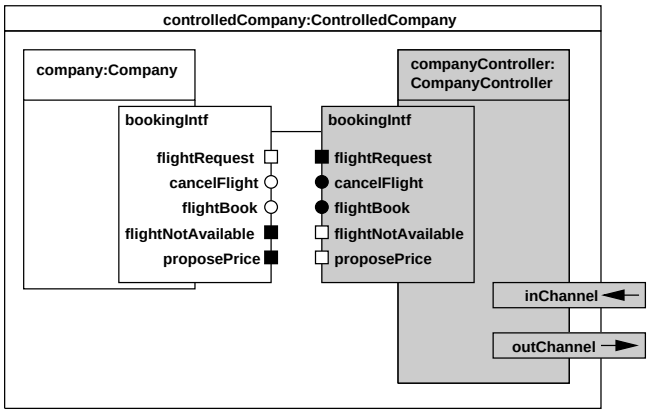

Fig. 8. Controlled Company

We choose to implement our proposal by following the second approach (see Fig. 8). In order to associate a protocol to a component at runtime, we use a single, complementary active entity that plays the role of a controller for the component (hence the name *controlled component* for the association of a bare component and a protocol). The role of a controller is to: (1) intercept the messages sent or received by the actual component and (2) decide whether these messages are either allowed or forbidden. This can be achieved by implementing the `logical state` pattern [20].

Fig. 9 depicts a simplified view of the `ComponentController` implementation class. The controller class implements the `Runnable` interface and the Java interface (`BookingIntfImported`) defining the required operations (`proposePrice` and `flightNotAvailable`) in the `bookingIntf` set.

The logical state pattern is implemented in two steps. First, all the possible states of the protocol are declared as private variables in the controller. Second,

```
public class CompanyController implements BookingIntfImported,
Runnable {
    private static final int ZERO = 0;
    private static final int ONE = 1;
    private static final int TWO = 2;
    private int controlState = ZERO; //Current State
    ...
    public void proposePrice(Price price) {...}
    public void flightNotAvailable() {...}
    ...
    public run(){
        ...
        while (true) { //No final state.
          switch (controlState) {
            case ZERO: ...; break;//Process flightRequestMessage
            case TWO:  ...; break;//Process flightBook & cancelFlight
          }
        }
    }
    ...
}
```

Fig. 9. CompanyController Code

the actions to be taken when the component can receive a request (according to the protocol state) are defined in the run method. The actions to be taken when the component can send a message are defined in the methods implementing the required operation of the component.

Fig. 10 presents the body of one of these methods (`proposePrice`). Two checks have to be realized before actually forwarding the message. The first check is related to the component current state. The second is the guard check. If the two checks are successful, the message is forwarded to the correspondent by employing a special entity called `channel` (see next subsection for details). If the message was correctly received, the protocol passes into a new state (state `TWO` as according to the definition of the STS) otherwise the execution blocks.

4.3 Channel Connections

In order to connect components into an architecture, we propose to employ channels [3]. A channel represents a one-to-one anonymous connection mechanism. It is also directed: messages flow from its *source* (where messages are sent) to its *sink* (where messages are received). Channels can be synchronous or asynchronous, mobile, with conditions, etc. In addition to its coordination role, a channel could be used in more sophisticated connection schemes. For example, a channel could integrate buffers or could be used to adapt one interface (and protocol) to another one. In the presented example we consider that all the components

```
public void proposePrice(Price price) {
  if (controlState != ONE) {//Protocol Compatibility Error}
  // guard is implicitelly true but BLOCK if false
  // Pack the actual message as: obj = new Object[]{...};

  try {
       writeChannel.write(obj);
       } catch (GuardException e) {
        //The guard of the correspondent is false. Block
       }
  controlState = TWO;//Transition successful. Change current state.
   }
```

Fig. 10. The `proposePrice` Implementation Code

are local and that the architecture is static. However, a channel could also be employed to connect remote component instances, possibly dynamically created at runtime.

To connect two interfaces we need two different channels. One channel is oriented from interface I1 to interface I2. The second channel is oriented form I2 to I1. Created inside the scope of the compound, the channel ends are transmitted as parameters to the communicating components (subcomponents) at instantiation time. The fact that the channels are not exported outside the composite scope ensures that they are exclusively used by the connected components.

We have created a class called `Channel`. This class implements two interfaces `WriteChannel` and `ReadChannel`. The `WriteChannel` interface defines the operation `write` related to the channel source. The `ReadChannel` interface defines the operations `read`, `commit`, and `cancel` related to the channel sink. The execution behaves differently depending on the communication type. If the communication is synchronous the sender blocks on the `write` method until the receiver reads (and commits or cancels, depending on the evaluation of the associated guard) the message. If the communication is asynchronous the `write` method blocks until the message is saved into a buffer, built inside the channel entity.

While the `Channel` class can be reused as is in many different connections, the controller classes have to be created for each bare component in the architecture. This can be done manually of course, but an automated solution is also possible. In fact, the language used to define the interfaces and the protocol of a component contains enough information to allow a tool to automatically create the required classes for an application. A pre-compiler based on the SableCC [19] framework has been developed. The pre-compiler takes as input the description of component (either primitive or compound) classes and generates the necessary controller classes.

5 Related Work

In the last decade, formal models with behavioural descriptions have been proposed in the context of software architectures [2, 24]. However, if they propose different analysis mechanisms for component architectures, they do not address the issue of taking protocols into account within the implementation, which are mandatory issues for seamless CBSE development processes.

In the concurrent object-oriented community, PROCOL [40] is one of the oldest proposals that deals with explicit protocols. PROCOL is a parallel C-based object-oriented language with communication based on one-way synchronous messages. Taking into consideration the sequential nature and the lack of protection against unscheduled accesses of object-oriented languages, the authors of PROCOL proposed to use synchronous message transfer in place of the RPC mechanism. This offers more potential for concurrency, and it may be seen as a restricted form of asynchronous communication (mailboxes being bounded to one message). The support language being an object-oriented one, required operations are not explicitly given. Moreover, receivers are explicitly given within the code. It is well known that component languages should rather explicit communications between components and try to separate communications from operation implementation and behavioural protocols. As far as the protocols are concerned, PROCOL relies on rational expressions extended with variables and guards. Each object is implemented as a process within the Unix environment and communicates under the control of a unique arbiter. Action sequencing is implemented by a nondeterministic finite automaton that is equivalent (in the accepting language sense) to the regular expression protocol of the object. Apart from the graphical presentation, PROCOL protocols have the same expressive power than STSs.

In [25], the authors present techniques to relate concurrent Java programs with a behavioural description given in the FSP (Finite State Processus) process algebra. Rather than really taking into account FSP protocols within a programming language, their goal is to be able to use FSP in a development process and to analyse models of threaded concurrent Java programs before coding them. This approach has been extended to the analysis of software architectures like [18] on web services where BPEL4WS is translated into FSP before analysis. The reverse approach is presented in [26] where the Darwin ADL has been used to address system construction using a Darwin to Corba IDL translation. However, behavioural protocols are not taken into account. The same holds for the different related approaches presented in [11].

In [17] the authors propose to integrate protocols within EJBs, at different levels. These protocols are given as labelled FSMs (Finite State Models) enriched by specific datatypes (for instance, to handle lists of allowed receivers) and related guards. A notion of consistency between the specified and the implemented protocol is also presented. Consistency is based on Nierstrasz work on regular types [29]. The main differences with our work are that we make it possible to use more complex datatypes within our protocols, we consider hierarchical models, and we address the development phase.

The SOFA [32] component model considers component types or templates. A component is an instance of a template. SOFA introduces the notion of interface, compound and primitive architectures as well as usual means to connect services between the subcomponents of an architecture. The description language introduces behavioural protocols and employs first-class connectors. Similarly to components, these connectors are described by a frame (required and provided interfaces) and an architecture (its implementation). The behaviour protocols are regular expressions denoting traces, *i.e.* sequences of events (required, provided, and internal calls). These protocols may be associated with interface, frame and architecture. [32] presents a model for the behaviour protocol-based description of hierarchical components. They define a notion of component substitution and a behavioural compliance between protocols. Protocol conformance can be verified at design time but protocols are not explicitly taken into consideration at implementation level.

6 Conclusions

We have presented in this paper a hierarchical component model supporting multiple interfaces with asynchronous and synchronous services. The main feature of this model is the introduction of explicit protocols based on Symbolic Transition Systems. This improves the capabilities of formal analysis of component systems with either model-checking or theorem proving. Then, we have presented a Java implementation of our model. A component corresponds to a controller encapsulating the STS part and a Java application provided with a well-defined component interface. The controller has the responsibility of intercepting communications and of triggering the right service on the inner Java code, depending on the state of the protocol. The communications are realized thanks to a notion of channel. This construction is well-known to provide benefits such as mobility, remote connections and reusability. In addition, channels are used in our implementation to ensure that a message arrives to its correspondent and also to notify the sender when the guard at the correspondent side is false.

A Component/Protocol Language Grammar

This appendix presents the grammare (in a BNF form) we defined in order to textually deal with components and their associated protocol.

A.1 Base Component

```
COMPONENT               ::= PRIMITIVE | COMPOUND
PRIMITIVE               ::= component name "{" component_body "}"
component_body          ::= interface_def+ ";" guards_def
                            ";" protocol_def ";"
interface_def           ::= interface name "{" operation_def+ "}"
operation_def           ::= op_type op_signature ";"

op_type                 ::= "PROVIDE" | "REQUIRE" | "REACT" |"NOTIFY"
op_signature            ::= type name "(" formal_parameter_list ")"

formal_parameter_list   ::= formal_parameter
                            | formal_parameter_list "," formal_parameter
formal_parameter        ::= type variable_declarator



guard_def               ::= guards "{" boolean_operation_def+ "}"
boolean_operation_def   ::= boolean_type op_signature ";"


protocol_def            ::= protocol_states protocol_transitions
protocol_states         ::= "states" "{" list_of_states "}" ";"
list_of_states          ::= state "(init)" | state | list_of_states "," state



protocol_transitions    ::= transitions "{" list_of_transitions "}" ";"
list_of_transitions     ::= (transition ";")+
transition              ::= state ":" operation_identifier guard "->" state

operation_identifier    ::= name | name "(" actual_parameter_list ")"
guard                   ::= @ | "[" boolean_expression "]"

actual_parameter_list   ::= actual_parameter
                            | actual_parameter_list "," actual_parameter
actual_parameter        ::= name | name "(" actual_parameter_list ")"

// The boolean expression in the guard is left associative and
//the precedence of operators is !, &&, || (from max to min)

boolean_expression      ::= boolean_expression or boolean_expression_and
                                    | boolean_expression_and
```

```
boolean_expression_and  ::= boolean_expression_and
                                  and boolean_expression_complement
                                   | boolean_expression_complement
boolean_expression_complement ::= complement boolean_expression_complement
                                   | boolean_expression_base
boolean_expression_base ::= operation_id
                                   | "(" boolean_expression ")"


complement              ::= "!"
and                     ::= "&&"
or                      ::= "||"
protocol                ::= "protocol"
states                  ::= "states"
transitions             ::= "transitions"
interface               ::= "interface"
state                   ::= identifier
variable_declarator     ::= identifier
name                    ::= identifier
type                    ::= identifier
boolean_type            ::= "boolean"
```

A.2 Composite Component

```
COMPOUND            ::= compound name "{" compound_body "}"
compound_body       ::= subcomponents_def+ connection_def+ interface_def+
subcomponent_def        ::= component_type name
connection_def      ::= connect name "." name "," name "." name ";"
interface_def       ::= interface name "{"  (export_def* import_def*) "}"
export_def          ::= export name "." name ";"
import_def          ::= import name "." name ";"

compound            ::= "compound"
connect             ::= "connect"
export              ::= "export"
import              ::= "import"
```

Acknowledgment. This work was partly supported by the ACI Sécurité Informatique, DISPO project.

References

1. M. Aiguier, F. Barbier, and P. Poizat. A Logic with Temporal Glue for Mixed Specifications. In *FOCLASA'2003*, volume 97 of *Electronic Notes in Theoretical Computer Science*, pages 155–174. Springer-Verlag, 2005.
2. R. J. Allen. *A formal approach to software architecture*. PhD thesis, Carnegie Mellon University, 1997.
3. F. Arbab, J. V. Guillen Scholten, F.S. de Boer, and M. M. Bonsangue. A channel-based coordination model for components. Technical report, Centruum voor Wiskunde en Informatica, 2002.
4. E. Astesiano, B.Krieg-Brückner, and H.-J. Kreowski, editors. *Algebraic Foundation of Systems Specification*. IFIP State-of-the-Art Reports. Springer-Verlag, 1999.
5. C. Attiogbé, P. Poizat, and G. Salaün. Integration of formal datatypes within state diagrams. In *FASE'2003*, volume 2621 of *Lecture Notes in Computer Science*, pages 344–355. Springer-Verlag, 2003.
6. L. Bellissard, S. B. Atallah, A. Kerbrat, and M. Riveill. Component-based programming and Application Management with Olan. In J. Briot, J. Geib, and A. Yonezawa, editors, *Object-Based Parallel And Distributed Computation*, volume 1107 of *LNCS*, pages 290–309. Springer-Verlag, Berlin, 1995.
7. J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.
8. C. Bigot, A. Faivre, J.-P. Gallois, A. Lapitre, D. Lugato, J.-Y. Pierron, and N. Rapin. Automatic Test Generation with AGATHA. In *TACAS'03*, volume 2619 of *Lecture Notes in Computer Science*, pages 591–596. Springer-Verlag, 2003.
9. B. Blanchet. From Secrecy to Authenticity in Security Protocols. In *SAS'02*, volume 2477 of *Lecture Notes in Computer Science*, pages 242–259. Springer-Verlag, 2002.
10. M. Calder, S. Maharaj, and C. Shankland. A Modal Logic for Full LOTOS Based on Symbolic Transition Systems. *The Computer Journal*, 45(1):55–61, 2002.
11. C. Canal. On the dynamic adaptation of component behaviour. In *WCAT'04*, pages 81–88, 2004.
12. C. Canal, J. M. Murillo, and P. Poizat, editors. *WCAT'2004 - Int. Workshop on Coordination and Adaptation Techniques for Software Entities*. 2004. Available at `http://wcat04.unex.es`.
13. S. Chiba. Load-time structural reflection in Java. In E. Bertino, editor, *Proceedings of the European Conference on Object-oriented Programming (ECOOP 2000)*, number 1850 in Lecture Notes in Computer Science, pages 313–336, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.
14. C. Choppy, P. Poizat, and J.-C. Royer. A global semantics for views. In T. Rus, editor, *International Conference, AMAST'2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer-Verlag, 2000.
15. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. From informal requirements to COOP: a Concurrent Automata Approach. In J.M. Wing and J. Woodcock and J. Davies, editor, *FM'99*, volume 1709 of *Lecture Notes in Computer Science*, pages 939–962. Springer-Verlag, 1999.

16. E. A. Emerson. *Temporal and Modal Logic*, volume B of *Handbook of Theoretical Computer Science*, chapter 16, pages 997–1072. Elsevier, 1990. J. Van Leeuwen, Editor.
17. A. Farías, Y.-G. Guéhéneuc, and M. Südholt. Integrating behavioral protocols in Enterprise Java Beans. In K. Baclawski and H. Kilov, editors, *Eleventh OOPSLA Workshop on Behavioral Semantics: Serving the Customer*, pages 80–89, 2002.
18. H. Foster, S. Uchitel, J., and J. Kramer. Model-based verification of web service compositions. In *ASE*, pages 152–163. IEEE Computer Society, 2003.
19. E. Gagnon. *SableCC, An Object-Oriented Compiler Framework*. PhD thesis, School of Computer Science McGill University, Montreal, November 1998.
20. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley Professional Computing Series. Addison Wesley, 1995. http://www.aw.com.
21. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98)*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer-Verlag, 1998.
22. A. Ingolfsdottir and H. Lin. *A Symbolic Approach to Value-passing Processes*, chapter 7 in [7], pages 427–478. Elsevier, 2001.
23. I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, 2003.
24. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of ESEC'95*, pages 137–53. IEEE, 1995.
25. J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, 1999.
26. J. Magee, A. Tseng, and J. Kramer. Composing distributed objects in CORBA. In *Proceedings of the Third International Symposium on Autonomous Decentralized Systems*, pages 257–63, Berlin, Germany, 9–11 1997. IEEE.
27. O. Maréchal, P. Poizat, and J.-C. Royer. Checking asynchronously communicating components using symbolic transition systems. In Z. Tari R. Meersman and al., editors, *CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1502–1519. Springer-Verlag, 2004.
28. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE - Transactions on Software Engineering*, 26(1):70–93, 2000.
29. O. Nierstrasz. Regular types for active objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 1–15. ACM Press, 1993.
30. J. Noyé, S. Pavel, and J.-C. Royer. A PVS Experiment with Asynchronous Communicating Components. In *17th Workshop on Algebraic Development Techniques*, Barcelona, Spain, 2004. `www.emn.fr/x-info/~jroyer/rrWADT04.pdf.gz`.
31. G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 329–400. Academic Press, August 1998.
32. F. Plasïl and S. Visnovsky. Behavior protocols for software components. *IEEE - Transactions on Software Engineering*, 28(11):1056–1076, November 2002.
33. P. Poizat and J.-C. Royer. Korrigan: a Formal ADL with Full Data Types and a Temporal Glue. Technical Report 83–2002, Laboratoire de Méthodes Informatiques, 2002. Available at `http://www.lami.univ-evry.fr/~poizat/publications-fr.php`.
34. P. Poizat, J.-C. Royer, and G. Salaün. Formal Methods for Component Description, Coordination and Adaptation. In *WCAT'04*, pages 89–100. Organizer position paper.

35. J.-C. Royer. Formal Specification and Temporal Proof Techniques for Mixed Systems. In *Proceedings of the 15th IPDPS 2001 Symposium, FMPPTA*, San Francisco, USA, 2001.
36. J.-C. Royer. The GAT approach to specify mixed systems. *Informatica*, 27(1):89–103, 2003.
37. J.-C. Royer. A Framework for the GAT Temporal Logic. In ISCA, editor, *Proceedings of the 13th IASSE'04 Conference*, 2004. ISBN: 1-880843-52-X.
38. J.-C. Royer and M. Xu. Analysing mailboxes of asynchronous communicating components. In D. C. Schmidt R. Meersman, Z. Tari and al., editors, *CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 1421–1438. Springer-Verlag, 2003.
39. K. J. Turner, editor. *Using Formal Description Techniques, An introduction to Estelle, LOTOS and SDL*. Wiley, 1993. ISBN 0-471-93455-0.
40. J. van den Bos and C. Laffra. PROCOL: A Parallel Object Language with Protocols. In Norman Meyrowitz, editor, *OOPSLA'89 Conference Proceedings*, pages 95–102. ACM Press, 1989.
41. D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.