Eclipse plugin for designing and developing Web Service orchestrations in JSCL *

Gianluigi Ferrari
Dipartimento di Informatica
Universita degli Studi di Pisa
Largo B. Pontecorvo, 3
I-56127 Pisa, Italy
giangi@di.unipi.it

Roberto Guanciale
IMT Lucca
Piazza S. Ponziano, 6
55100 Lucca, Italy
roberto.guanciale@imtlucca.it

Daniele Strollo
Dipartimento di Informatica
Universita degli Studi di Pisa
Largo B. Pontecorvo, 3
I-56127 Pisa, Italy
strollo@di.unipi.it

ABSTRACT

We introduce a semantic-based development environment that includes support for designing and implementing coordination policies of service oriented applications using the event notification paradigm. Our environment is implemented as an Eclipse-plugin. In this paper, we describe the basic facilities of the environment of and illustrate its use in the development of a service-based automotive application.

Categories and Subject Descriptors

D.1.3 [**Programming Techiniques**]: Concurrent Programming—*Distributed programming*

; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering, Object-oriented design methods*

; D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments*

General Terms

Web Services, WS-Choreography, Eclipse

Keywords

Event Notification, SOA, Choreography, Eclipse plugin

1. INTRODUCTION

The *event notification* paradigm is a frequently used paradigm for programming/modeling distributed system. In event notification, systems are built through distributed components that interact as *publishers* and/or *subscribers*. A component, to request a service from other ones, raises an *event* that can activate a reaction for *subscribers* that previously *subscribed* for such kind of events.

The JSCL is a middleware based on formal semantics adhering to the event notification paradigm. As always with any middleware, good supports are needed to program and drive the implementation. The key feature of JSCL is the strict interplay between foundational aspects and the implementation. In particular the development of service oriented applications could benefit greatly from an environment to reason about properties of coordination policies. We have developed an environment (*JSCL4Eclipse*) to treat these aspects. The environment consists of a Eclipse plugin that supports the development of applications based on top of JSCL. The chosen Eclipse platform permits to reuse the facilities of the existing framework to the JSCL platform. Moreover the environment can be extended using the standard Eclipse methodology. The environment provide a graphical editor for an high level designing of event based coordination policies.

We present the prototype implementation of the environment that supports the design and execution of JSCL based applications. We are investigating the integration of model checking techniques to provide a property proving tool to the environment. We are also studying aspects that can be treat using static analisys based on type checking.

This paper is organized as follows: in Section 2 we recall the main concept of JSCL and event notification paradigm. The Section 3 presents the *JSCL4Eclipse* framework. the graphical notation and a simple example. Finally, in Section 4 we present how model and implement the SENSORIA [7] car repair scenario using our framework.

2. PRELIMINARIES

The programming model we present here is a adaptation of the event notification paradigm (EN for short) where components (in the following referred also with the term *services*) interact through the raising of signals for notifying event happened during their execution. The initial step of our work is a process calculus, the Signal Calculus (SC), presented in [8], that defines the basic primitives useful to deal with components interacting in a signal event based fashion. From the distilling of the main constructs introduced in the calculus we have isolated the concepts useful to describe component interfaces, in the means of their entry points and of their connections. Furthermore, we have implemented a middleware, Java Signal Core Layer (JSCL), as a set of Java API that reflect the spirit of the ideas exposed

*(Produces the permission block, and copyright information). For use with SIG-ALTERNATE.CLS. Supported by ACM.

in SC. Several other implementations can be adopted and inter operate with JSCL components, assuming that they adhere on the communication protocol suggested in SC and on the signal serialization.

In JSCL, known concepts of event based paradigms are considered in a distributed and open environment where involved components can be allocated on different execution sites and can join existing execution of other components. Differently from the standard event notification paradigm, in our proposal, we define a full decentralized system, namely no intermediate service is demanded to implement the publication/subscription of events and of their related notifications. In literature [12] this pattern is often referred as *non brokered*, in contrast with the brokered one that implements publish subscribe mechanism on top of classification of signals without taking into account the involved components. Essentially, brokered solutions can be compared to *linda spaces* [9] and the operations required for registering and dispatching messages are intermediated by a central service, namely the *broker*, which is responsible to keep track of the components interested to receive such messages. In our solution, instead, each subscriber explicitly defines the class of events it is interested to and the set of components from which it is interested to receive notifications. This mechanism will be clarified in the following when we introduce the concept of links connecting publishers and subscribers among them.

Even if JSCL represents an 'easy to use' light weight framework for programming distributed components, we have developed an Eclipse plugin to simplify the designing of connections among components and to automatically generate the related code to execute.

2.1 Java Signal Core Layer

Java Signal Core Layer is a framework implementing choreography of *services* in an event notification style. In this section we only focus on the main aspects of JSCL strictly related to the EN paradigm, while, in Section 2.2, it is provided an overview of the primitives of the framework that, later, in Section 2.3, are used to program components.

The event notifications are delivered to involved components through *signals* defined as messages encapsulating the description of the raised events. According to EN paradigm, events are classified into *topics* that classify homogeneous events. Moreover, depending on the topics, signals are enriched with additional information useful to retrieve the data needed to their handling. In JSCL the set of information contained into signals is split in two parts, the *signal type* and the *session data*. The *signal type* declares the topic of the signal while the *session data* contains the *actual* carried information. Signals in JSCL are *not persistent*, meaning that, once the notification for an event has been delivered to the proper target components, the corresponding signal is removed from the system leaving no track. This property is mandatory to keep the distribution of the connection among components and their management. Moreover, for similar reasons, the notification dispatching is implemented in *non anonymous* way, namely for each notification it is always possible to retrieve the component on which the event happened and the last component from which the signal has been forwarded.

Component interfaces are defined in terms of *reactions* and *links*. *Reactions*, in JSCL implemented with the *input port* class, are the linguistic device for implementing the subscription mechanism, while links provide the capability to implement publication. Reactions are defined in the form $(topic, task)$, and install a binding between signal topics and the behavioral tasks to activate at the reception of corresponding signals. Tasks activated by signal triggering are executed as threads inside the component providing so, through a shared state space, a mechanism for interacting among them. Once installed a reaction for topic t on a component b, the subscription for notifications raised by the component a is obtained by creating a link $l = (a, b, t)$. *Links* are so described as tuples defining the source and the target component and the signal topic that can be carried on them. The link creation primitive exposed in JSCL can be invoked outward the components by an external entity similarly to the subscription delegation mechanism of the EN paradigm. Notice that, in our implementation, links have not been explicitly defined, their handling is demanded to publishers in order to distribute the network topology information among involved services.

2.2 JSCL programming interface

To abstract from the particular underlaying network adopted, JSCL introduces a lower architectural layer, the Inter Object Communication Layer (`iocl`) that defines the minimal structure a network must expose to allow the distribution and the interaction of components. At this layer, they are defined the primitives for creating, publishing and retrieving distributed components. The `iocl` acts in behalf of the network hiding to the programmer the mechanism used to exchange messages among components. Moreover, the middleware allows several instances of `iocl` to coexist, using the addressing mechanism to identify the network infrastructures. By introducing protocol *prefixes* in fact, the addresses themselves identify the particular instance of `iocl` to adopt. For example we use *socket* notation as address prefix to indicate components interacting via usual socket connections, as well we use *mem* or *xsoap* prefixes to indicate respectively, in memory and xsoap (an implementation of SOAP Web Services) plugins. Since component addressing is strictly related to the particular `iocl` adopted, the address creation and manipulation is demanded to the proper `iocl` plugin. In order to simplify the development, JSCL defines proxy artifacts that hide the communication primitives needed to activate remote executions that strictly depend on the network technology adopted by the `iocl`, providing to the programmer the usual object oriented facilities.

The `iocl` implements the primitives for creating, publishing and retrieving components. The methods `createAddress` and `createComponent` create a new instance of a service address and of a JSCL component, respectively. The methods `publish` and `getComponent` are used to locally publish the component on the `iocl` and to retrieve a remote component instantiating a proxy that locally exposes the component remote interface. Moreover, a link for topic t can be created by invoking the method `iocl.createLink(topic, sourceAddr, destAddr)`, where *sourceAddr* and *destAddr* are the addresses expressed with the `iocl` addressing form.

Once instantiated a component it is possible to attach to it the reactions. This step is performed by invoking the method *addInputPort* on the component instance[1]. The input port is created by invoking the constructor *InputPort (topic, task)*. *Task* is obtained by implementing the *SignalHandlerTask* abstract class. Essentially it is required to implement the method *handle (Signal signal)*. The middleware is responsible to retrieve at the reception of a signal, the component reaction to activate and to run its handling method in a thread space passing to it the signal instance. The handling tasks may act on the internal state of the components and on the parameters carried with the received signal. Signals can be managed through the methods *setType(type)*, *getParameter(key)*, *setParameter(key, value)* whose meaning is obvious. Since signals can represent data messages exchanged among services to coordinate their tasks, it is useful to retrieve from each signal the flow session they belong to, this can be done by invoking the method `signal.ID()`. Notice that a new unique session identifier is automatically assigned to a signal instance at instantiation time. Having reactions encapsulated into components, the related tasks can access all the capabilities offered by their containers (*signal emission*, *reaction update*, etc.). For creating links on a component for outgoing signals it is possible to use the method `component.createLink(topic, targetAddress)` that is an alias for the primitive exposed by the `iocl`. The *emitterAddr* parameter is now omitted since the component itself is the sender of notifications. Through the method *emit (signal, async)* it is possible to forward outside a notification, the second parameter specifies if the notification must be sent in asynchronous manner and can be omitted assuming its value to *true* by default. The emission will happen regardless of the component (if any) that will actually receive the emitted signals. It might even be no target component is connected; the middleware will correctly deliver signals transparently to the signal emitter.

2.3 A programming example

In this section we introduce a small example to highlight the main features of JSCL. In Section 3.3 the same example will be used to illustrate the facilities offered by the graphical toolkit.

We consider to have a scenario composed by three services, respectively a, b and c. We remark that the meta name description that we adopt here, corresponds to an unique URL to which services are bound. Moreover, we consider to treat only two classes of events in our application, respectively evt_1 and evt_2. The service a represents the entry point of the whole application and, autonomously, at a certain point, it sends a notification for both event topics. The components b and c are two generic services able to react to events with topic evt_1 and to events with topics evt_1, evt_2, respectively. The connection topology is the following: a is connected for event topics evt_1 to b and for event topics evt_2 to c, while b is connected to c for events with topic evt_1.

Since in JSCL topics are represented as integers, we use the values 1001 and 1002 to deal with evt_1 and evt_2 topics, respectively. For simplicity we assume both services be alive and their reactions already bound before a starts to send notifications.

The steps that must be performed to implement the proposed application are the following:

(a) for each component, at initialization phase, a new `iocl` plugin is instantiated (we suppose to use an xsoap implementation)

```
IOCLPluginLoader loader =
 new IOCLPluginLoader (new IOCLAddress
  ("net.tao4ws.jtl.core.IOCL.XSoap.IOCLImpl"));
IOCLPluginI factory=(IOCLPluginI)loader.getIOCLInstance();
```

(b) components are *instantiated* and *published* by using the `iocl` primitives previously defined

```
ComponentAddress compAddr =
 factory.createAddress ("http://localhost:9093/a");
GenericComponent a = factory.createComponent(compAddr);
factory.registerComponent(a);
```

(c) reactions are bound to components by executing

```
c.addInputPort(new SignalInputPort(
 1001, // signal topic to handle
 new SignalHandlerTask() {
  // behavior
  public Object handle (Signal s){
   // ... manage signal
   emit (s);
   return null;
  }
}));
```

(d) once components have been instantiated and published, and once their reactions have been bound, links among components can be created as explained below:

```
// 1001 is the signal topic
// compSrcAddr and CompDestAddr are supposed to be
// the source and target peers addresses, respectively.
factory.createLink(1001, compSrcAddr, compDestAddr);
```

We remark that, the algorithms exposed, are just intended as snippets useful to give an idea of how to program with JSCL. Notice that some simplifications have been done, for example the link creation can also be executed on the service itself as previously discussed. In our assumption, the three services are distributed on different hosts so the code exposed before must be executed on the machines on which each service is hosted. Only the link creation primitive can be executed on a different host respect the involved services.

2.4 Additional programming facilities

Another key feature of JSCL, is the ability of defining orchestrated services, allowing so to define services obtained by coordination of other services. Externally an orchestrated service keeps the same characteristics of usual JSCL components, moreover it exposes two additional methods *registerEntryPoint* and *registerEndPoint*. Both methods require two parameters, the *topic* handled and the *address* of component to add as entry or exit point. Entry and exit points are generic components that declare externally the interface of the whole orchestration. While it is possible to attach only an entry point to an orchestration, several exit points can be defined. The input ports installed on the entry point define the reactions activable on the orchestration while once an orchestrated service is linked to other components, its exit points are automatically linked to these external services.

Other *value added* of JSCL are the *logical ports* and the *guards* on links. The logical ports permit to define first order

[1]Notice that the component instance referred here is the local component created by the `iocl` and not the proxy artifact remotely bound.

logic based applications offering, moreover, an easy mapping to flow-chart diagrams. In JSCL they are defined the *and*, *or* and *not* logical ports. Logical ports are instantiated by declaring two kinds of topics corresponding to the boolean *true* and *false* values. Respect to general components, logical ports are able to handle only signals corresponding to signal topics associated to the boolean values. This assumption restricts the use of primitives for signals firing and input port creation to this topic subtype. To apply the boolean relations to the outgoing signals of several components it is sufficient to invoke the method `insertComponent`, meaning that on the output links of notifier it will be applied a filtering. Obviously the output links of the inserted component must be compatible with the types handled by the port. Differently from usual logical relations, the ports can have several components connected both in input and in output. Essentially the logical ports have been introduced for simplifying the design of service composition. In first instance these components permit to the developer an easy view of the connections and furthermore they can be used in conjunction with the other primitives in order to express more complex patterns. For example, referring to the *workflow patterns* presented in [16], we can describe the *parallel split* and the *synchronization* between two components a and b by adding an *and* port filtering they outgoings. In the same manner the *or* port can be adopted for implementing the *exclusive choice*.

Link guards give the possibility to attach, during the creation of a link, a guard that must be avaluated before sending the signal. If, and only if, the evaluation of the guard is *true* the signal is sent through the link otherwise it is removed from the system. Guards are built by extending the abstract class `Guard` implementing the method `processSignal(signal)`. For example, if we want to be sure to avoid the delivering of signals having the session parameter `creditCard` we can attach to links the guard described below:

```
new Guard() {
 public boolean processSignal (Signal s){
  if (signal.getSession().
    getParameter("creditCard") == null)
   return true;
  return false;
 }
};
```

3. JSCL4ECLIPSE

In this section we introduce *JSCL4Eclipse*, a programming framework, developed in the form of eclipse [3] plugin that supports the designing and programming of JSCL applications. Our framework is composed by three layers: an *editor* that permits to graphically model service coordination, a *model transformation* that *compiles* a model into Java code and JSCL *middleware* as runtime support. The plugin has been implemented using GMF [4], that provides a generative infrastructure for developing editors based on EMF [5] as model representation and GEF [6] as graphical support.

The development chain of JSCL based applications is constituted of three steps (Figure 1). At first it is graphically defined the model of the coordination. This model is subsequently used to generate the Java code on top of JSCL API. Finally it must be implemented the internal logic of the

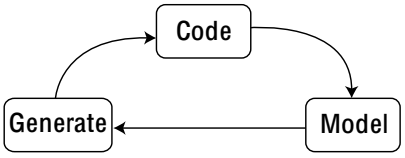
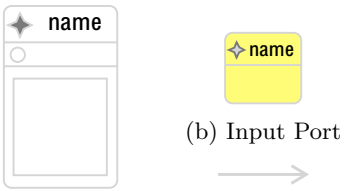


Figure 1: JSCL development chain



(a) Component (b) Input Port (c) Link

Figure 2: JSCL diagram elements

generated components. Now we present the main concepts of the graphical editor and the *code generation* of JSCL *applications*.

3.1 Graphical Notation

The JSCL graphical notation captures the sequence of activities via the description of the network topology of involved components. This notation has some relations with *BPMN* [14]. The main difference is that *BPMN* directly defines the flow of messages exchanged among components using a flow chart based notation. JSCL notation, instead, defines the correlation among services, while the message sequence depends by the component internal behavior and is not directly caught at design time.

The diagram elements of JSCL graphical notation are presented in Figure 2. In JSCL services are implemented through components. JSCL components can represent BPMN *pools* on the assumption that services are used to implement the participants. JSCL input ports can be interpreted as BPMN activities, because they are the handlers of messages and implement *atomic business logics*. Similarly to BPMN activities, which can be depicted into BPMN pools, input ports must be encapsulated inside JSCL components. Links in BPMN define the activity sequence, while, in JSCL, they define publishers/subscribers relationships. Notice that a JSCL diagram cannot contain events. In fact events are raised internally from components, and links specify to which components deliver events according with their topics. Finally the JSCL graphical notation does not explicitly formalize the notion of BPMN gateways. Moreover these elements can be represented by components with a *pre-defined internal logic* (e.g. the logical ports defined in Section 2.4).

The example presented in Section 2.3 has been depicted in Figure 3 using our graphical notation. The collaboration is composed by three services. The first one, a, has no input port, since it does not react to any event. Component b has one input port which handles signals of type evt_1. Finally the component c has two input ports: *P1* and *P2* handling evt_1 and evt_2 topics, respectively. Notice that an

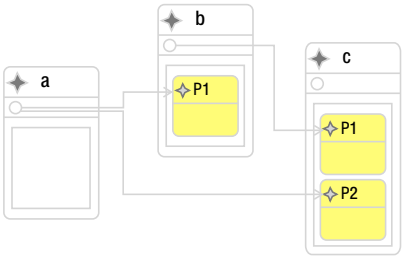


Figure 3: JSCL diagram of the example

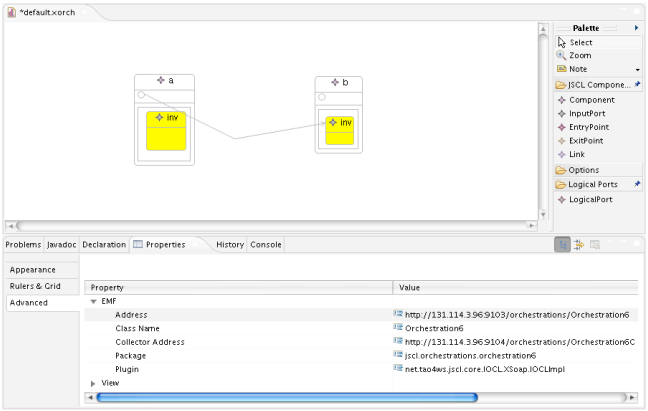


Figure 4: JSCL4Eclipse diagram editor

event raised by a can activate a reaction inside the component c in two ways: if the signal has type evt_2, it is directly delivered to c activating its P_2 input port. Otherwise the signal can activate the b reaction, that, on its turn, can notify an event, with type evt_1, to c.

3.2 Basic Facilities

We describe how to model and program service coordination using our framework. As described above our programming environment is integrated into Eclipse. Starting from an eclipse project, the environments permits to create a new JSCL model using the eclipse *project wizard*, selecting *JSCL model* as type and providing an unique name. The plugin creates into the project a *xscl* and a *xorch* files that are used to store the model and graphical information, respectively. Opening the *xorch* file, eclipse starts the default editor associated to this extension: the JSCL *diagram editor* (see Figure 4). A new blank sheet is displayed. It contains, on the right side, a toolbar with the set of primitives that can be used. On the bottom side is presented the property view. This contextual view permits to examine and modify properties of the selected element.

New primitives can be added to the choreography by dragging them to the sheet. As depicted in Figure 3, some primitives are *nested*, namely they can be drawn only into an existing one (e.g. input ports can be defined only inside components).

The selection of a primitive updates the property view. Primitive properties can be classified into three kinds: *model properties* (e.g. the *unique name* for components and the *type of signal* for input ports), *diagram properties* (e.g. primitive position in the sheet and text notes) and *generation properties* (e.g. *class name* for the generated code and *url* for components). Notice that components have two special generation properties: *generate source code* and *publish*. These features are useful in the following conditions:

- when it is required to generate the whole code for components and nested input ports (enabling both the properties)
- when it is required to create a new instance of an already implemented component (enabling only the *publish* property)
- when it is required to use in the coordination an already remotely published component (disabling both properties)

At the end of the modeling phase, it is possible to generate the source code. This is performed via the contextual menu of the *xscl* file. According with the *generation properties* specified in the diagram, the source code is generated as follows:

- A *orchestrator class* is generated. This class have a static *main* method that creates the coordination. The orchestrator instantiates each component and, if their *publish property* is true, it publishes them to the proper addresses, otherwise it obtains a remote proxy for them. Then the orchestrator creates links among components (we remark that JSCL allows to remotely crate links).
- For each component having *generate source property* enabled, a *component class* is created. This class creates the input ports of the component and represents a shared state among input ports.
- For each input port a class is created. This class implements the handler for messages of the specified type. To implement the internal *business logic*, the body of the handle method must be filled. This method will be invoked upon the reception of signals having the suitable type.

Each generated class allows for each method to specify the *generated* annotation. Source code with this annotation will be overridden by further code generations. To prevent this behavior the annotation must be disabled, informing the environment that the source code has been specialized. The environment supports a simple navigation mechanism: starting from the diagram view, the contextual menu *Edit code* opens in the default Eclipse Java editor the corresponding source file.

3.3 Example

In order to exemplify our framework, we present how to model and implement the example presented in Section 2.3. The first step is to create a new coordination, using the *new JSCL Diagram* wizard. Initially the package and the url of the coordination must be specified. These properties

File	Functionality
Orchestration.java	Instantiates the three components, publishes them and creates the links among components.
ComponentA.java ComponentB.java ComponentC.java	Instantiate for components the input ports as handlers according for signal types described in the model
InputPortB1.java InputPortB2.java InputPortC.java	Define the *handle* methods that will be invoked upon the reception of signals having the proper type

Table 1: Files generated for the example

represent the package where source code will be generated and a address to use as prefix for nested components.

The three components can be added by dragging three times the component primitive into the sheet. Then, they must be named a, b and c respectively through the property view. For this example other properties are not mandatory. one input port for the component b and two for the component c must be added and their properties changed such that the port of b and one port of c react to the same signal type (i.e. evt_1) and the other port of c reacts to a different signal type (i.e. evt_2). Three links must be added to complete the model: the first link from the component a to the input port owned by b, the second one from a to the input port of c than handle signal type evt_2 and the last link from b to the evt_1 input port of c. Notice that the signal types for these links must not be specified, having previously configured the signal types to which each input port refers.

To generate the source code, all components must have enabled the *generate* and *publish* properties. Then, using the contextual menu it is possible to generate the source code. In Table 1 are listed the generated files and their functionalities. Notice that the framework assumes some default values. For example, if the *class* and the *package name* properties for an input port are not specified, a default value is automatically assigned according with the values of the owner component.

Once the source code has been generated, it is possible to implement the component internal logic. The handle method of the three input ports can be implemented as follows:

```
/* @Generated=false */
protected class InputPortC1 extends SignalHandlerTask {
  public Object handle (Signal s) {
    emit(s);
  }
}

/* @Generated=false */
protected class InputPortB1 extends SignalHandlerTask {
  public Object handle (Signal s) {
    System.out.println("Commit_Received");
  }
}

/* @Generated=false */
protected class InputPortB2 extends SignalHandlerTask {
  public Object handle (Signal s) {
    System.out.println("Rollback_Received");
  }
}
```

To ensure that the framework will not overwrite these methods, the *generated* method annotation must be changed to *false*.

3.4 A reusable component

Since the framework provides the way to use existing class as component implementation, it is useful to define some standard components that can be used several times in one ore more choreographies. Is such case the framework generates a main choreography class that creates new instances of these components, using the previously defined implementing classes. One component that can be useful is a sort of *logical and*. Interpreting a signal type as a logical true and another one as logical false, this component act as follows: it waits two notifications of the same event instance (the two signal received have the same ID). If both received signals have the true type, the component emits one signal having true type. If at least one received signal have false type, the component emits one false signal. Notice that in all cases the emission of a signal can be performed only after the reception of two signals with the same ID. The And component implementation is depicted in Code 1.

```
class AndComponent extends GenericComponent {
  protected Map state = new HashMap();
  protected trueSiglal = 1001;
  protected falseSignal = 1002;
  protected class AndPort extends SignalHandlerTask {
    public Object handle (Signal s){
      syncronized(s.ID()) {
        if (!state.containsKey(s.ID()))
          state.set(s.ID(), s.getType());
        else if (state.get(s.ID()) == signalTrue &&
                 s.getType() == signalTrue) {
            emit(s);
            state.remove(s.ID());
        }
        else {
            s.setType(signalFalse);
            emit(s);
            state.remove(s.ID());
        }
      }
    }
  }
  AndComponent() {
    this.addInputPort(
        new SignalInputPort(signalTrue, new AndPort())
    );
    this.addInputPort(
        new SignalInputPort(signalFalse, new AndPort())
    );
  }
}
```

Code 1: And component implementation

This component can be used in a JSCL diagram. For example, to synchronize the activities of two components that can deliver signals having type corresponding to the logical true. The AndComponent can be reused in a diagram by setting for each instance of the component the *generate code* property to false.

4. A CASE STUDY

In this section we explain how the SENSORIA car repair scenario [15] can be developed using JSCL framework. A car manufacturer offers a service that once a users car breaks down the car system tries to locate a garage, a tow truck and a rental car service so that the car is towed to the next garage and repaired meanwhile the car owner may continue his travel. The interdependencies between the service bookings make it necessary to have an orchestration with compensations. Before any service lookup is made, the credit card is charged with a security amount. Before looking for a tow truck, a garage must be found as it poses additional constraints to the candidate tow trucks. If finding a tow truck fails, the garage appointment must be revoked. If renting a car succeeds and finding either a tow truck or a garage appointment fails, the car rental must be redirected to the

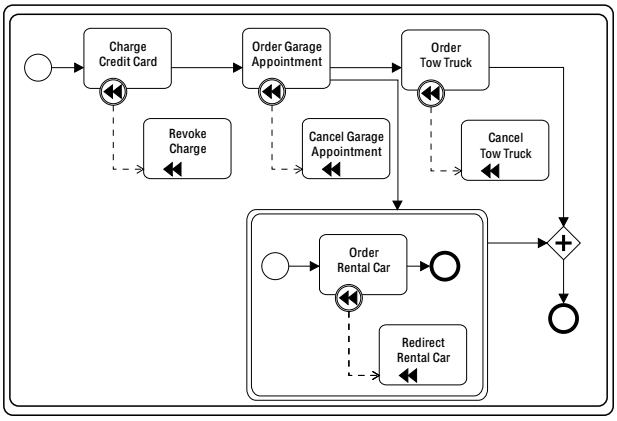


Figure 5: Car repapair scenario BPMN model

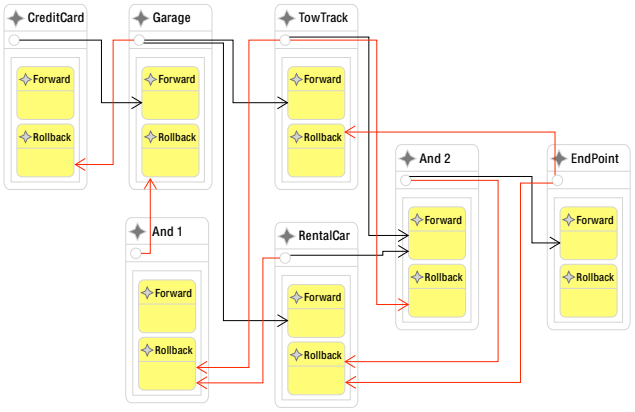


Figure 6: Car repapair scenario JSCL model

broken down car's actual location. If the car rental fails, it should not affect the tow truck and garage appointment.

The BPMN model of this scenario is presented in Figure 5. Notice that the model exploits the transactional and compensation facilities of BPMN and that the car rental service is a subtransaction, since it does not affect other activities.

4.1 Designing the Car Repair Scenario

To model the car repair scenario as a JSCL diagram we assume that each participant is represented by a JSCL component. We use two types of signals: *forward* and *rollback*. The first event type is used to inform a component that all the previous activities have been completed, while the second one is used to inform a component that an exception has been occurred, and that all following activities have completed their compensation. According with the signal types defined, each component have two input ports: one that handles forward signals, executing the corresponding main activity, and the other one that handles rollback signals, executing the corresponding compensation if the main activity has been previously completed without errors.

In Figure 6 the JSCL model of the car repair scenario is presented. For clarity, we use the black color for forward links and red for the other ones. The model contains two instances of the AndComponent defined in Section 3.4. In this case we consider the forward events as true and rollback events as false. The first AndComponent is used to synchronize the backward flow before the Garage compensation. The second AndComponent has two issues: to synchronize the forward flow, and to execute the compensation of RentalCar if both the OrderTowTruck fails and the RentalCar main activity has been completed successfully. Notice that, according with the scenario, the OrderTowTruck compensation is not executed if the RentalCar fails. To reuse the previously defined AndComponent the *publish* property of the two instances must be enabled, while the *generate code* property not. The EndPoint component represents the BPMN final state and allows to reuse this coordination to other ones as a subtransaction. This component simply forward received signals, without change their type.

Before generating the source code, the diagram properties must be set as in Section 3.3 (e.g. component publication urls and coordination package).

4.2 The Car Repair Scenario implementation

Since an activity compensation must be executed only if the corresponding activity has been previously completed without errors, components must keep track of successful completed activities. To implement this functionality each component declares a public Map, indexed on signal ID (see Code 2).

The internal logic of a component is implemented through its two input ports: the forward port and the rollback one. A forward input port (e.g. Code 3) tries to execute the corresponding activity. If the execution does not rise an exception, the input port updates the state of the component and emits a forward signal, informing connected components that the main activity has been successfully completed. If an error occurs, the input port emits a rollback signal. A rollback input port (e.g. Code 4) checks that the corresponding activity has been completed and then executes the compensation.

Since the car rental activity is a sub-transaction, a local error has no side effects. To implement this, the forward port of the car rental component (see Code 5) always emits a forward signal, regardless the successful of the activity.

```
public class ComponentTruck extends GenericComponent {
  public Map state = new HashMap();
  ...
```

Code 2: The Truck component

5. CONCLUDING REMARKS

The Signal Calculus extension, presented in [10], takes advantage of topics for defining an algebraic structure of events for implementing more complex logics directly encoded inside the signal type (e.g. logical ports, join and non deterministic choice). We are implementing this concepts in JSCL, providing a foundational description of the behavior of standard components that represent BPMN gateways. Another important feature exposed by JSCL is the *dynamic*

```
protected class ForwardTruck extends SignalHandlerTask {
  public Object handle (Signal s) {
    try {
      TruckComponent parent = (TruckComponent)getParent();
      // ToDo: Program here the truck activity
      parent.state.set(s.ID(), true);
      emit(s);
    }
    catch (Exception e) {
      s.setType(ROLLBACK);
      emit(s);
    }
  }
}
```

Code 3: The Truck forward input port

```
protected class RollbackTruck extends SignalHandlerTask {
  public Object handle (Signal s) {
    TruckComponent parent = (TruckComponent)getParent();
    if (parent.state.get(s.ID())) {
      // ToDo: Program here the truck compensation
    }
    parent.state.remove(s.ID());
    emit(s);
  }
}
```

Code 4: The Truck rollback input port

binding of reactions that permits to components to be independent from the support languages used to implement them. We have presented the prototype implementation of JSCL environment. Our long term research goal is to build a semantic based programming environment:

- exploits type notion ([10]) to characterize coordination properties
- exploits a spatial logic [2] to characterize structural properties of JSCL diagrams. We want to extend our framework with model checking techniques to prove properties during the development phase.

We are studying other Eclipse tools (e.g.[11, 1]) that provide verification and analysis capabilities for Java program specifications. In particular, JML is a behavioral interface specification language that provides a variety of useful constructs for specifying Java assertions, pre/post-conditions, class invariants, and light-weight semantic constraints. Finally we should provide foundational based transformation from BMPN models to JSCL diagrams and implementations, and we would use BPMN as behavior specification language for the verification phase.

6. REFERENCES

[1] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *Electr. Notes Theor. Comput. Sci.*, 80, 2003.

[2] V. Ciancia and G. Ferrari. Co-algebraic models for quantitative spatial logics. *QAPL07. Fifth Workshop on Quantitative Aspects of Programming Languages.*, 2007. To appear.

[3] Eclipse Foundation. Eclipse - an open development platform. Technical report. `http://www.eclipse.org`.

[4] Eclipse Foundation. Eclipse Graphical Modeling Framework. Technical report. `http://www.eclipse.org/gmf/`.


```
protected class ForwardCar extends SignalHandlerTask {
  public Object handle (Signal s) {
    try {
      CarComponent parent = (CarComponent)getParent();
      // ToDo: Program here the truck activity
      parent.state.set(s.ID(), true);
    }
    catch (Exception e) {
    }
    emit(s);
  }
}
```

Code 5: The CarRental forward input port

[5] Eclipse Foundation. Eclipse Modeling Framework. Technical report. `http://www.eclipse.org/modeling/emf/`.

[6] Eclipse Foundation. Graphical Editing Framework. Technical report. `http://www.eclipse.org/modeling/gef/`.

[7] EU-FP6. Software Engineering for Service-Oriented Overlay Computers. Technical report. `http://www.sensoria-ist.eu/`.

[8] G. L. Ferrari, R. Guanciale, and D. Strollo. Jscl: A middleware for service coordination. In Najm et al. [13], pages 46–60.

[9] D. Gelernter. Generative communications in Linda. *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112, January 1985.

[10] D. S. Gianluigi Ferrari, Roberto Guanciale and E. Tuosto. Coordination via types in an event-based framework. *FORTE'07. 27th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems*, 2007. Submitted.

[11] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In T. Ball and S. K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003.

[12] Y. Liu and B. Plale. Survey of publish subscribe event systems. Technical Report TR574, Computer Science Department, Indiana University, 2003.

[13] E. Najm, J.-F. Pradat-Peyre, and V. Donzeau-Gouge, editors. *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006.*, volume 4229 of *Lecture Notes in Computer Science*. Springer, 2006.

[14] Object Management Group. Business Process Modeling Notation. Technical report. `http://www.bpmn.org/`.

[15] M. Wirsing, A. Clark, S. Gilmore, M. M. Hölzl, A. Knapp, N. Koch, and A. Schroeder. Semantic-based development of service-oriented systems. In Najm et al. [13], pages 24–45.

[16] P. Wohed, W. M. van der Aalst, M. Dumas, and A. H. ter Hofstede. Pattern Based Analysis of BPEL4WS. Technical report, Department of Computer and Systems Sciences Stockholm University/The Royal Institute of Technology, Sweden, nov 2003.