From Architectural to Behavioural Specification of Services[1]

Laura Bocchi[2] José Luiz Fiadeiro[3]

Department of Computer Science, University of Leicester, UK

Alessandro Lapadula[4] Rosario Pugliese[5] Francesco Tiezzi[6]

Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, Italy

Abstract

Many efforts are currently devoted to provide software developers with methods and techniques that can endow service-oriented computing with systematic and accountable engineering practices. To this purpose, a number of languages and calculi have been proposed within the SENSORIA project that address different levels of abstraction of the software engineering process. Here, we report on two such languages and the way they can be formally related within an integrated approach that can lead to verifiable development of service components from more abstract architectural models of business activities.

Keywords: Service-Oriented Computing, Service Component Architecture, modelling languages, formal methods, process calculi

1 Introduction

Service-Oriented Computing (SOC) is an emerging paradigm that aims to support a new generation of software applications that can run over globally available computational network infrastructures where they can procure services on the fly (subject to a negotiation of service level agreements – SLAs) and bind to them so that, collectively, they can fulfil given business goals. One of the many efforts that are currently devoted to support SOC is directed to establishing methodologies and sound engineering approaches that allow software developers to move from ad-hoc to systematic and accountable engineering practices. Therefore, a number of languages and formalisms are being investigated within the FET

[1] This work has been supported by the EU project SENSORIA, IST-2 005-016004.

[2] Email: bocchi@mcs.le.ac.uk

[3] Email: jose@mcs.le.ac.uk

[4] Email: lapadula@dsi.unifi.it

[5] Email: pugliese@dsi.unifi.it

[6] Email: tiezzi@dsi.unifi.it

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Global Computing integrated project Sensoria [2] to address different levels of abstraction of the software engineering process.

In this paper, we report on the way two such languages can be formally related within an integrated approach that can lead to verifiable development of service components from abstract architectural models of business activities. None of these languages is 'complete' in the sense that none addresses all aspects of SOC. Rather, they result from a deliberate decision to select key issues of the paradigm that can be investigated and tested individually and brought together once they are well understood.

The languages we consider address modelling aspects that arise at different levels of abstraction. On the one hand, SRML (the Sensoria *Reference Modelling Language* [14]) offers primitives for modelling composite services and business activities that abstract from the actual process of discovery, selection, binding, reconfiguration and session management. This process is assumed to be provided by the underlying middleware and, as such, is not part of the modelling activity, which allows the designer to concentrate on the business aspects of services. On the other hand, COWS (*Calculus for Orchestration of Web Services* [20]) is a process calculus for specifying and combining service-oriented systems that addresses a lower level of abstraction where the dynamic aspects of SOC need to be explicitly modelled. Its design has been inspired by well-known process calculi as well as the OASIS standard language for orchestration of web services WS-BPEL [26]. In fact, COWS can model and handle distinctive features of (web) services such as correlation-based communication, compensation activities, service instances and interactions among them, race conditions among service instances and service definitions, inter alia.

The objective of relating the two languages is precisely to provide an operational semantics for SRML by making explicit in COWS some of the run-time aspects that SRML abstracts from. In fact, the semantics that we have provided for SRML (e.g., [3,15]) is declarative in the sense that it relies on mathematical domains (configuration graphs and state transition systems) to make precise the meaning of its different constructs. Through the implementation in COWS we get an operational semantics that can reveal the requirements that these constructs put on the underlying 'middleware' with the advantage that COWS is still one level of abstraction above actual web service languages and platforms.

From a technical point of view, the main challenge is in providing an implementation that is modular in the structure of SRML models (i.e., the structure of the COWS term that implements a SRML module follows the structure of the module itself). This aspect, which we call the 'architecture' of the implementation, is one of the main technical aspects that we discuss in the paper, especially the way it reflects the methodology of software development that we are building around SRML. We are currently developing a software application for automatising the implementation, which will also pave the way for the analysis of SRML models by exploiting the reasoning mechanisms and verification techniques that are being made available for COWS. These include a type system to check confidentiality properties [19], a stochastic extension to enable quantitative reasoning on service behaviours [28], a static analysis to establish properties of the flow of information between services [5], and a logic and a model checker to express and check functional properties of services [13]. This is an important advantage over related approaches (see Section 6).

The rest of the paper is organised as follows. Section 2 provides a survey of SRML and COWS. Section 3 presents the case study that is used throughout the paper. Section 4 describes the architecture of the implementation. Section 5 presents the implementation

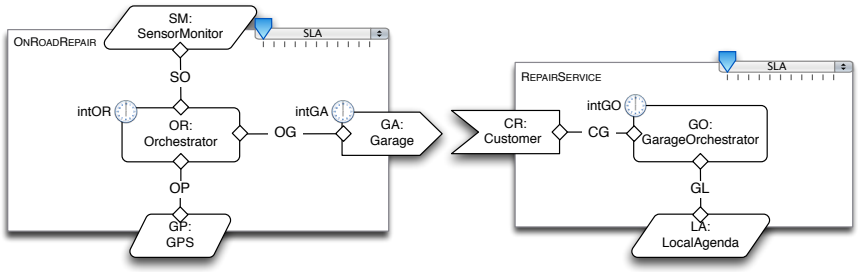

Fig. 1. Activity module OnRoadRepair (left-hand side) and service module RepairService (right-hand side)

through the case study. Section 6 concludes by discussing pointers for current/future work.

2 A glimpse of SRML and COWS

This section presents a survey of SRML and COWS. The overview of SRML gives a high-level description of the aspects captured by its modelling primitives. This is done over a scenario selected from an automotive case study being developed in Sensoria. Due to lack of space, the overview of COWS gives only a glimpse of its semantics, a full account of which can be found in [20].

An Overview of SRML. SRML provides primitives for modelling service-oriented applications whose business logic involves the orchestration of interactions among more elementary components — typically provided locally and bound at design-time — and the invocation of services provided by external parties, discovered and selected at run-time.

SRML is inspired by SCA (*Service Component Architecture* [23]) and is independent of the languages and platforms that are currently being provided for web [4] (or grid [16]) services. An encoding of WS-BPEL is available that illustrates how SRML (static) models can be (partially) implemented in more concrete languages [8].

To illustrate and discuss the use of the language and methodology, we chose a reference scenario, depicted in Fig. 1, that involves an activity OnRoadRepair that takes place in a software system (embedded in a vehicle) handling engine failures detected by a sensor. When the activity is triggered, the system (1) determines the current location of the car by using a GPS device, and (2) binds to a repair service selected among those offered by nearby garages that can ensure best levels of assistance, including a tow truck if necessary.

In SRML the unit of design is what we call *module*. There are two kinds of modules. *Activity* modules specify applications developed to satisfy the requirements of a specific business organisation and not to be published as a service. An example is the activity OnRoadRepair that will have been developed by, or for, the car manufacturer. *Service* modules are developed (by, or for, service providers) to be published in repositories in ways that allow them to be discovered when a request for an external service is published in the run-time environment. An example is the repair service that OnRoadRepair will procure when the engine-failure sensor is activated.

A module is specified in terms of a number of entities and the way they are interconnected. For example, the activity module OnRoadRepair shown in Fig. 1 (left-hand side) involves the following software entities: SM (the sensor that triggers the activity), GP (the GPS system), and OR (the orchestrator that coordinates the interactions with the external

services and GP). These entities are interconnected through *wires*, each of which defines an interaction protocol between two entities. Typically, wires deal with the heterogeneity of partners involved in the activity by performing data integration, which is useful when, for instance, a car has to travel across different countries. OnRoadRepair relies on an external service (i.e., GA) for booking a garage and calling a tow-truck, the discovery of which will be triggered, on-the-fly, according to the conditions detected by the sensor.

As illustrated, every activity module declares interfaces of four possible kinds: (1) one and only one *serves-interface* that binds the activity to the application that triggered its execution (e.g., SM on the left-hand side of Fig. 1), (2) a number of *uses-interfaces* (possibly none) representing entities that are shared among different activity instances and persist to the life-cycle of each single instance (e.g., GP on the left-hand side of Fig. 1), (3) a number of *component-interfaces* (at least one) that bind to components that are created when the activity is launched (e.g., OR on the left-hand side of Fig. 1), (4) a number of *requires-interfaces* (possibly none) that bind the activity to services that are procured externally when certain conditions become true (e.g., GA on the left-hand side of Fig. 1).

Service modules such as RepairService in Fig. 1 (right-hand side) provide a service to the external environment and can be dynamically discovered and invoked (instead of being launched directly by users). Compared with activity modules, they have one *provides-interface* — CR in the example — instead of a serves-interface.

Notice that the workflow of a module is defined collectively by the components in its configuration and the wires that connect them, which facilitates modular development and reuse driven by the structure of the business domain. SRML does not support a hierarchical definition of modules (e.g., refining a component as a module).

All interfaces involve a signature declaring the set of supported interactions and a specification of the behaviour associated with them. See [14] for details on the formalisms used for specification (basically, temporal logic and state machines). In Section 3 we provide the necessary details of the specification to understand the implementation over COWS.

SRML also offers primitives for defining internal and external configuration policies. The internal policies (indicated by clocks) define the initialisation and termination conditions of each component and the conditions that trigger the discovery process of each external service. For instance, intGA in Fig. 1 is the condition that triggers the discovery of GA; it is defined in terms of the events that can occur during the execution of OnRoadRepair. The external policies (indicated by the rulers) express constraints for Service Level Agreements (SLA). For this purpose, SRML adopts the c-semiring approach to constraint satisfaction and optimisation developed in [6].

An Overview of COWS. COWS is a formalism for specifying and combining services that has been influenced by the principles underlying WS-BPEL. It provides a novel combination of constructs and features borrowed from well-known calculi such as non-binding receiving activities, asynchronous communication, polyadic synchronization, pattern matching, protection, and delimited receiving and killing activities. These features make it easier to model service instances with shared states, processes playing more than one partner role, and stateful sessions made by several correlated service interactions, inter alia.

The syntax of COWS is presented in Table 1. It is parameterized by three countable and pairwise disjoint sets: the set of *(killer) labels* (ranged over by $k, k', \ldots$), the set of *values* (ranged over by v, v', ...) and the set of 'write once' *variables* (ranged over by x,

s	::=	$u \bullet u'!\bar{\epsilon} \mid g$	(invoke, receive-guarded choice)
	$\mid$	$[e]\, s \mid s \mid s \mid * s$	(delimitation, parallel composition, replication)
	$\mid$	$\mathbf{kill}(k) \mid \{\!\{s\}\!\}$	(kill, protection)
g	::=	$\mathbf{0} \mid p \bullet o?\bar{w}.s \mid g + g$	(empty, receive prefixing, choice)

Table 1
COWS syntax

$y, \ldots$). The set of values is left unspecified; however, we assume that it includes the set of *names*, ranged over by n, m, o, p, ..., mainly used to represent partners and operations. The syntax of *expressions*, ranged over by ϵ, is deliberately omitted; we just assume that they contain, at least, values and variables, but do not include killer labels (that, hence, can *not* be exchanged in communication).

We use w to range over values and variables, u to range over names and variables, and e to range over *elements*, i.e. killer labels, names and variables. Notation $\bar{\cdot}$ is used for tuples (ordered sequences) of homogeneous elements, e.g. $\bar{x}$ is a compact notation for denoting the tuple of variables $\langle x_1, \ldots, x_n \rangle$ (with $n \geq 0$). We assume that variables in the same tuple are pairwise distinct. We adopt the following conventions for operators' precedence: monadic operators bind more tightly than parallel, and prefixing more tightly than choice. We omit trailing occurrences of $\mathbf{0}$, writing $p \bullet o?\bar{w}$ instead of $p \bullet o?\bar{w}.\mathbf{0}$, and write $[e_1, \ldots, e_n]\, s$ in place of $[e_1] \ldots [e_n]\, s$. Finally, we write $I \triangleq s$ to assign a name I to the term s.

Invoke and *receive* are the basic communication activities provided by COWS. Besides input parameters and sent values, both activities indicate an *endpoint*, i.e. a pair composed of a partner name p and of an operation name o, through which communication should occur. An endpoint $p \bullet o$ can be interpreted as a specific implementation of operation o provided by the service identified by the logic name p. An invoke $p \bullet o!\bar{\epsilon}$ can proceed as soon as the evaluation of the expressions $\bar{\epsilon}$ in its argument returns the corresponding values. A receive $p \bullet o?\bar{w}.s$ offers an invocable operation o along a given partner name p. Execution of a receive within a *choice* permits to take a decision between alternative behaviours. Partner and operation names are dealt with as values and, as such, can be exchanged in communication (although dynamically received names cannot form the endpoints used to receive further invocations). This makes it easier to model many service interaction and reconfiguration patterns.

The *delimitation* operator is the *only* binder of the calculus: $[e]\, s$ binds e in the scope s. Differently from the scope of names and variables, that of killer labels cannot be extended (indeed, killer labels are not communicable values). Delimitation can be used to generate 'fresh' private names (like the restriction operator of the π-calculus [25]) and to delimit the field of action of kill activities. Execution of a *kill* activity $\mathbf{kill}(k)$ causes termination of all parallel terms inside the enclosing $[k]$, which stops the killing effect. Critical activities can be protected from the effect of a forced termination by using the *protection* operator $\{\!\{s\}\!\}$.

Delimitation can also be used to regulate the range of application of the substitution generated by an inter-service communication. This takes place when the arguments of a receive and of a concurrent invoke along the same endpoint match and causes each variable argument of the receive to be replaced by the corresponding value argument of the invoke within the whole scope of variable's declaration. In fact, to enable parallel terms to share the state (or part of it), receive activities in COWS do *not* bind variables (which is different from most process calculi).

Execution of *parallel* terms is interleaved, except when a kill activity or a communication can be performed. Indeed, the former must be executed *eagerly* while the latter must ensure that, if more than one matching receive is ready to process a given invoke, only one of the receives with greater priority (i.e. the receives that generate the substitution with 'smaller' domain, see [20] for further details) is allowed to progress. Finally, the *replication* operator $*\, s$ permits to spawn in parallel as many copies of s as necessary. This, for example, is exploited to model persistent services, i.e. services which can create multiple instances to serve several requests simultaneously.

3 Specification of an Automotive Case Study

The graphical notation used in Section 2 to specify the automotive case study has the advantage of being intuitive and facilitating the identification of the relationships among the involved entities. However, it abstracts from a number of details that need to be accounted for when defining an implementation. For this reason, we have defined a detailed and 'tractable' textual notation also for SRML, the Backus-Naur Form syntax of which is given in [7]. Here, we present the specification of the case study using this textual notation.

Table 2 presents an excerpt of the specification of the module *OnRoadRepair* illustrated in Fig. 1. *OnRoadRepair* is defined by a number of component/serves/requires-interfaces and their associated type (e.g., *OR* of type *Orchestrator*). We do not include uses-interfaces because we did not defined their implementation in COWS yet. The types of interfaces (**SPECIFICATIONS**) are defined below. The internal policies **init** and **term** of *OR* define the initialisation and termination conditions of the component. Initially, the local variable s has value *INIT*. The component is compulsorily terminated when either the final state is reached (i.e. $s = FINAL$) or a fatal error occurs (i.e. $s = ERR$). According to the internal policy **trigger** of *GA* the discovery process is triggered by the condition $s = READY$.

The wires *SO* and *OG* connect pairs of nodes by defining a relationship between the interactions and the parameters of the corresponding specifications.

Each specification is composed by a syntactical interface (**INTERACTIONS**). In SRML interactions are asynchronous and can be one-way (i.e., receive **rcv** or send **snd**) or conversational (i.e., receive-and-send **r&s**, or send-and-receive **s&r**). A number of interaction events is associated with each conversational interaction: an initiation event (denoted by 🔔), a reply-event (denoted by ✉), and so on. Interactions can involve a number of parameters for each phase of the conversation (e.g., 🔔-parameters for the initiation, ✉-parameters for the reply, etc.). One-way interactions have associated only one 🔔-event.

Every instance of *Orchestrator* can engage in the interactions *init* and *bookGarage*. The former is of type **rcv** and permits to receive data from the sensor monitor installed in the car. The data are represented by the parameter *data* of type *carData*. The interaction *bookGarage* is used for engaging with a garage service. This interaction is conversational (of type **s&r**) and has one 🔔-parameter *data* and one ✉-parameter *price* through which the price for repairing the car can be obtained. In the initial state, i.e. when $s = INIT$, an *Orchestrator* can perform only the transition *data_receiving*, which is triggered by the event *init*🔔 and changes the internal state (as usual, we denote by s' and $data'$ the next value of the local state variables s and *data*). The transition *reqToGarage* has no trigger and is executed as soon as the guard $s = READY$ is true. The transition sends the event *bookGarage*🔔 and assigns the sensor data (stored in the local variable *data*) to the parameter *bookGarage.data*.

```
MODULE OnRoadRepair is
  COMPONENTS OR : Orchestrator init s = INIT term s = FINAL ∨ s = ERR
  SERVES SM : SensorMonitor
  REQUIRES GA : Garage trigger s = READY
           ...
  EXTERNAL POLICY carUserSLAconstraints
  WIRES SO : SM OR activation ↔ init : 🔔 sensorData ↔ data
        OG : OR GA bookGarage ↔ acceptBooking : 🔔 data ↔ info,
                                                ✉ price ↔ servicePrice
        ...
  SPECIFICATIONS
    BUSINESS ROLE Orchestrator is
      INTERACTIONS
          rcv init 🔔 data : carData
          s&r bookGarage 🔔 data : carData
                         ✉ price : moneyVal
          ...
      ORCHESTRATION
        local s : [INIT, READY, WAITING, GA_PRICE, ..., FINAL, ERR],
              data : carData, much : moneyVal, ...
        transition data_receiving
          triggeredBy init🔔
          guardedBy s = INIT
          effects s′ = READY ∧ data′ = init.data

        transition reqToGarage
          guardedBy s = READY
          effects s′ = WAITING
          sends bookGarage.data = data ∧ bookGarage🔔

        transition respFromGarage
          triggeredBy bookGarage✉
          guardedBy s = WAITING
          effects s′ = GA_PRICE ∧ much′ = bookGarage.price
        ...
    LAYER PROTOCOL SensorMonitor is
      INTERACTIONS snd activation 🔔 sensorData : carData
      BEHAVIOUR SensorMonitorBehaviour

    BUSINESS PROTOCOL Garage is
      INTERACTIONS r&s acceptBooking 🔔 info : carData
                                     ✉ servicePrice : moneyVal
      BEHAVIOUR GarageBehaviour
```

Table 2
The textual definition of the module *OnRoadRepair*

The event is sent to the (dynamically discovered) garage service. Finally, by means of transition *respFromGarage*, the price required by the garage service can be received and stored in the local variable *much*.

An excerpt of the specification of the module *RepairService* is shown in Table 3. It contains the component *GO* (of type *GarageOrchestrator*) connected to the provides-interface *CR* (of type *Customer*) by the wire *CG*. The *GarageOrchestrator* provides the interaction *handleRequest* of type **r&s**, which is made available through the provides-interface to bind to customers upon selection (e.g. *bookGarage*). The interaction *handleRequest* can be en-

```
MODULE RepairService is
  COMPONENTS GO : GarageOrchestrator init s = INIT term s = FINAL
  PROVIDES CR : Customer
  REQUIRES ...
  EXTERNAL POLICY garageSLAconstraints
  WIRES CG : CR GO   getRequest ↔ handleRequest : 🔔 dataFromCar ↔ d,
                                                  ✉ cost ↔ c
       ...
  SPECIFICATIONS
    BUSINESS ROLE GarageOrchestrator is
      INTERACTIONS
          r&s handleRequest   🔔 d : carData
                              ✉ c : moneyVal
          ...
      ORCHESTRATION
        local s : [INIT, HANDLING, ..., FINAL], data : carData
        transition reqResp
          triggeredBy handleRequest 🔔
          guardedBy s = INIT
          effects s' = HANDLING ∧ data' = handleRequest.d
          sends handleRequest.c = computePrice(data') ∧ handleRequest ✉
        ...
    BUSINESS PROTOCOL Customer is
      INTERACTIONS s&r getRequest   🔔 dataFromCar : carData
                                    ✉ cost : moneyVal
      BEHAVIOUR CustomerBehaviour
```

Table 3
The textual definition of the module *RepairService*

gaged by executing the transition *reqResp*. In this way, the data of the customer's car are received and processed to calculate the cost of the repair (through *computePrice*(·)), after which the computed cost is sent back to the customer.

Notably, the textual notation considered above is parameterized by an unspecified set of Service Level Agreement constraints (**EXTERNAL POLICY**), and by an unspecified set of service descriptions (**BEHAVIOUR**) that represent the behavioural specifications of abstract references (i.e., requires-interfaces in SRML).

4 Modular Architecture of the Implementation

From an operational point of view, a SRML module cannot be considered as an isolated entity; its role needs to be understood in relation to the middleware through which discovery and binding are ensured and the environment of services that are available over the network. This section discusses how the elements that compose a SRML configuration can be defined in terms of an orchestrated system in COWS. We illustrate our approach by means of the automotive case study introduced in the previous sections.

To make the implementation modular, the SRML configuration modelling the automotive case study is decomposed in a number of areas of concern, numbered one to six in Fig. 2:

(1) *Creation of an activity or service instance.* Every implementation of a SRML module is intended as a *factory* (1a) that handles the creation of different instances. Each instance

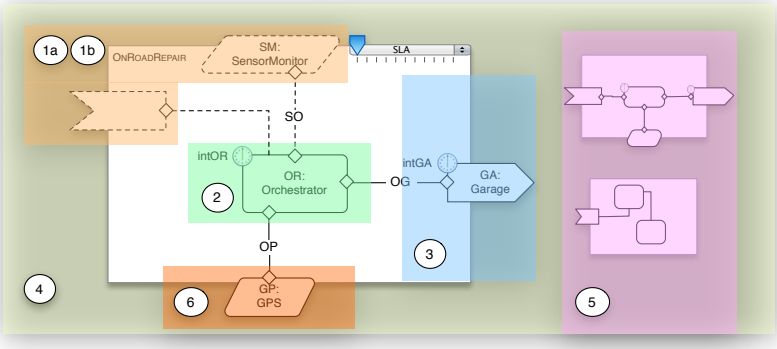

Fig. 2. Decomposition of OnRoadRepair into areas of concern

of a module has an associated *instance handler* (1b) that implements message correlation and maps the interaction/parameter names of the interface to those of the correct components of the module.

(2) *Orchestration.* The orchestration consists of the executable pattern of interactions described by the set of components internal to the SRML module.

(3) *Discovery of a service.* To bind new service components to those in the instance that triggered the discovery, we need what we call a *discovery handler*. From a module's perspective, the information for handling the process of discovery of each of its requires interfaces includes (1) a specification of the required syntactic/behavioural properties (i.e., the business protocol), (2) a specification of the SLA constraints given by the external policies and (3) the condition that triggers the discovery process (i.e., the trigger condition associated with a specific requires-interface in SRML). The discovery handler of a module includes a *requires handler* for each requires-interface of the module. A requires handler implements the mapping of names and parameters of a specific requires-interface to those of the components of the discovered module as established by the wires.

(4) *Middleware.* It consists of those functionalities that support the execution of SRML configurations. Among other things, the middleware enables the discovery and binding processes by relying on a *broker* — a discovery and reasoner entity that selects the most suitable service that matches a given requires-interface among those stored in a *repository*. The middleware also includes a *matchmaking agent* supporting the matching of functional descriptions and a *constraint solver* supporting the negotiation of Quality of Service properties. This is where COWS offers a layer of abstraction that is still above that of a dedicated middleware, thus allowing us to 'parametrise' the implementation and remain independent of specific technologies. For instance, web service architectures currently provide only very limited brokerage facilities via the technology UDDI [4].

(5) *Environment.* It consists of the activities and services published in some repository.

(6) *Bottom layer.* It consists of the set of persistent entities, which typically already exist when a service instance is created and which may be shared among different instances (e.g., GP of type GPS in OnRoadRepair).

According to this architecture, the COWS representation of a service module is

$$Module^{(1,2,3)} \mid Middleware^{4} \mid Environment^{5} \mid BottomLayer^{6}$$

where $Module^{(1,2,3)}$ is of the form:

$$Factory^{1a}.(InstanceHandler^{1b} \mid Orchestration^{2} \mid DiscoveryHandler^{3})$$

The superscripts establish a correspondence between the terms and the parts of a SRML configuration illustrated in Fig 2. One advantage of this architecture is that it permits an incremental development of the different aspects of the implementation.

5 Semantics of the Reconfiguration

In this section, we give a flavour of the implementation of SRML in COWS through the case study introduced in Sections 2 and 4. A more complete account of the implementation can be found in the following technical report [7]. Firstly, we present the static aspects of the implementation, i.e. how a SRML configuration is implemented in COWS, and then the dynamic ones, by showing the COWS term resulting from a reconfiguration.

Static aspects of the implementation. The COWS term representing all the entities involved in the automotive case study, where $\langle\!\langle \cdot \rangle\!\rangle$ represents the implementation in COWS of the enclosed term, is

$$\langle\!\langle \textbf{MODULE } OnRoadRepair \textbf{ is } \ldots \rangle\!\rangle \mid \langle\!\langle \textbf{MODULE } RepairService \textbf{ is } \ldots \rangle\!\rangle \mid Middleware \mid Environment \mid BottomLayer$$

where *Middleware* is the term (*Broker* | *Registry* | *ConstraintSolver* | *MatchmakingAgent* | ...), while *Environment* contains, at least, a COWS term representing the car's sensor monitor that interacts with the module instance through the serves-interface. The term *BottomLayer* is left unspecified since the implementation of the bindings performed through uses-interfaces is in progress.

The car's sensor monitor can be represented by the following COWS term:

$$[id_{sm}]\,(\, OnRoadRepair \bullet create!\langle sensorMonitor, id_{sm}\rangle \mid OnRoadRepair \bullet activation!\langle id_{sm}, \text{🔔}, \text{“}gps = (4348.1143N, 1114.7206E), fuelPr = 60psi, brakeBias = 70/30, \ldots\text{”}\rangle\,)$$

This term directly invokes the service factory of the module *OnRoadRepair* without resorting to a discovery mechanism (recall that *OnRoadRepair* is an activity module). The operation *create* does not correspond to an interaction supported by the original SRML module but to the factory of the COWS implementation of *OnRoadRepair*. It has the effect of creating a new instance of the module and initialising it with the sensor monitor partner name *sensorMonitor* and the fresh instance identifier id_{sm}. In parallel, the sensor monitor sends the collected data by invoking the COWS operation corresponding to the interaction *activation* provided by the interface *SM* of *OnRoadRepair*.

A SRML module corresponds to a persistent COWS service that can be instantiated by invoking the operation *create* with the partner name of the module (that coincides with the name of the module, as e.g., *RepairService*). We assume that names of modules are distinct; this is reasonable because, at the real implementation level, module partner names can be thought of as URIs.

The implementation of *RepairService* is:

$$Broker \bullet pub!\langle RepairService, \text{“}Customer\ \texttt{is}\ \ldots\text{”}, garageSLAconstraints\rangle$$
$$\mid\ *\,[x_{cust}, x_{ext\_id}]\, RepairService \bullet create?\langle x_{cust}, x_{ext\_id}\rangle.$$
$$[id_{intra}]\,(\,ProvidesInt \mid RequiresInt \mid Wires \mid Components\,)$$

With respect to the architecture of the implementation of a service module we have seen in Section 4, we have that *Factory* corresponds to the replicated receive along the endpoint $RepairService \bullet create$, while *InstanceHandler*, *Orchestration* and *DiscoveryHandler* correspond to *ProvidesInt*, *Wires* | *Components* and *RequiresInt*, respectively.

The implementation of the module *OnRoadRepair* is similar, except for the absence of the publication activity (i.e. the invoke along the endpoint $Broker \bullet pub$) and the replacement of *ProvidesInt* with the term *ServesInt* implementing the serves-interface *SM*.

To instantiate a module, a service has to provide its partner name (to allow the created instance to reply) and a conversation identifier (stored in x_{ext_id}) that will be used for correlating inter-module communication to avoid interference among instances of the same module. To guarantee absence of interference during intra-module communication when a new module instance is created, a fresh conversation identifier id_{intra} is generated. This identifier is necessary because communication among entities of an instance (i.e. components, wires and interfaces) are performed along the same endpoints used by other instances of the same module. The intra-module identifier differs from the external identifier to prevent external entities from directly contacting internal entities. Such an identifier is also used in the communication with *Broker* during the discovery phase.

The implementation of a wire is a persistent COWS service that catches a send event (by means of a receive activity) from a connected entity, adapts the communication endpoint and forwards the adapted event (by means of an invoke activity) to the other entity. For example, the wire *OG* between *OR* and *GA* in *OnRoadRepair* is:

$$*\,[x_{data}]\, OG_{roleA} \bullet bookGarage?\langle id_i, 🔔, x_{data}\rangle.\, GA \bullet acceptBooking!\langle id_i, 🔔, x_{data}\rangle$$
$$\mid\ *\,[x_{servicePrice}]\, OG_{roleB} \bullet acceptBooking?\langle id_i, ✉, x_{servicePrice}\rangle.$$
$$OR \bullet bookGarage!\langle id_i, ✉, x_{servicePrice}\rangle$$

The term above uses two distinguished partner names to interact with the connected entities: the partner name OG_{roleA} is used to catch messages from the left end of the wire, while OG_{roleB} is used for the right end (see the specification of *OG* in Table 2). Notably, id_i is the conversation identifier for intra-module communication of the *OnRoadRepair*'s instance.

An instance of a module can interact with instances of other service modules only after the successful completion of the discovery phase. In particular, when a requires-interface of the considered instance is triggered, it starts the discovery process by interacting with *Broker*. Consider, for example, the requires-interface *GA* of *OnRoadRepair*. After its activation, it sends a message with the business protocol *Garage* and the external policy *carUserSLAconstraints* to *Broker*. Then, *MatchmakingAgent* and *ConstraintSolver* execute a matchmaking process between the pair (“*Garage* `is` ...”, *carUserSLAconstraints*) and the pairs of business protocols and SLA constraints stored in *Registry*. If matching succeeds, *Broker* sends back to *GA* a message with binding information.

The implementation of *GA* is as follows:

$$
\begin{array}{l}
GA \bullet trigger?\langle id_i \rangle. \\
(\, Broker \bullet disc!\langle OnRoadRepair, id_i, \text{“}Garage\ \texttt{is}\ \ldots\text{”}, carUserSLAconstraints \rangle \\
\quad \mid [x_p, x_{acceptBooking}]\, OnRoadRepair \bullet GA?\langle id_i, x_p, x_{acceptBooking} \rangle. \\
\qquad [id_{ext}]\, (\, x_p \bullet create!\langle OnRoadRepair, id_{ext} \rangle \\
\qquad\quad \mid x_p \bullet bindingInfo!\langle id_{ext}, acceptBookingResp \rangle \\
\qquad\quad \mid * [x_{info}]\, GA \bullet acceptBooking?\langle id_i, \triangle, x_{info} \rangle. \\
\qquad\qquad (\, x_p \bullet x_{acceptBooking}!\langle id_{ext}, \triangle, x_{info} \rangle \\
\qquad\qquad\quad \mid [x_{servicePrice}]\, OnRoadRepair \bullet acceptBookingResp?\langle id_{ext}, \boxtimes, x_{servicePrice} \rangle. \\
\qquad\qquad\qquad\qquad OG_{roleB} \bullet acceptBooking!\langle id_i, \boxtimes, x_{servicePrice} \rangle\,) \\
\qquad\quad \mid \ldots\,)\,)
\end{array}
$$

where id_i is the conversation identifier for the intra-module communication of the considered *OnRoadRepair*'s instance. The discovery process is triggered by a signal along the endpoint $GA \bullet trigger$, which is sent by the implementation of the component *OR* when the instance state is set to *READY* by transition *data_receiving*.

An instance of a service module can receive messages from the customer service that has created it by means of a provides-interface. For example, the implementation of the provides-interface *CR* of *RepairService* is

$$
\begin{array}{l}
[x_{getRequest}]\, RepairService \bullet bindingInfo?\langle x_{ext\_id}, x_{getRequest} \rangle. \\
\quad * [x_{dataFromCar}]\, RepairService \bullet getRequest?\langle x_{ext\_id}, \triangle, x_{dataFromCar} \rangle. \\
\qquad (\, CG_{roleA} \bullet getRequest!\langle id_{intra}, \triangle, x_{dataFromCar} \rangle \\
\qquad\quad \mid [x_{cost}]\, CR \bullet getRequest?\langle id_{intra}, \boxtimes, x_{cost} \rangle.\ x_{cust} \bullet x_{getRequest}!\langle x_{ext\_id}, \boxtimes, x_{cost} \rangle\,)
\end{array}
$$

The implementation of a provides-interface is symmetric to that of a requires-interface, i.e. it replaces the external identifier within an incoming message with the internal identifier. Notice that, in case of conversational interactions, to allow a provides-interface to reply to the corresponding requires-interface, the latter has to send to the former some binding information (e.g., in case of *GA*, the operation name *acceptBookingResp*).

Due to lack of space, we do not show here the implementation of components. It suffices to know that a component is implemented by a COWS term that performs invoke/receive activities corresponding to SRML interactions according to the types of the interactions and the orchestration logic of the component.

Dynamic aspects of the implementation. Suppose now that the COWS service implementing *RepairService* has already been published in the *Broker*'s registry. This means that it has already communicated to *Broker* its partner name, the business protocol of its provides-interface, and its external policy, by performing the invoke activity $Broker \bullet pub!\langle RepairService, \text{“}Customer\ \texttt{is}\ \ldots\text{”}, garageSLAconstraints \rangle$. Suppose also that the sensor monitor has already contacted, and instantiated, the module *OnRoadRepair* by invoking operation *create*, and that the created instance has performed transition *data_receiving*. A possible evolution of this scenario is described below.

(1) *OnRoadRepair triggers the process of discovery and binding.*

(i) Execution of transition *data_receiving* of *OnRoadRepair* has set the state to *READY*. Thus, the triggering condition of its requires-interface *GA* holds true and, hence, the implementation of *GA* starts the discov-

ery process. Assume that the broker, through *MatchmakingAgent* and *ConstraintSolver*, selects the pair ("*Customer* **is** ...", *garageSLAconstraints*) published in the repository by *RepairService* as the best match for the pair ("*Garage* **is** ...", *carUserSLAconstraints*) sent by *GA*. Then, *Broker* returns the message $\langle id_i, RepairService, getRequest\rangle$ along the endpoint $OnRoadRepair \cdot GA$. Therefore, x_p is replaced by the partner name *RepairService*, and $x_{acceptBooking}$ by *getRequest*. This way, the implementation of *GA* evolves into the following term:

$$
\begin{aligned}
&[id_{ext}]\,(\,RepairService \bullet create!\langle OnRoadRepair, id_{ext}\rangle \\
&\quad \mid RepairService \bullet bindingInfo!\langle id_{ext}, acceptBookingResp\rangle \\
&\quad \mid *\,[x_{info}]\,GA \bullet acceptBooking?\langle id_i, \text{🔔}, x_{info}\rangle. \\
&\qquad (\,RepairService \bullet getRequest!\langle id_{ext}, \text{🔔}, x_{info}\rangle \\
&\qquad\quad \mid [x_{servicePrice}] \\
&\qquad\quad\; OnRoadRepair \bullet acceptBookingResp?\langle id_{ext}, \text{✉}, x_{servicePrice}\rangle. \\
&\qquad\quad\; OG_{roleB} \bullet acceptBooking!\langle id_i, \text{✉}, x_{servicePrice}\rangle\,) \\
&\quad \mid \ldots\,)
\end{aligned}
$$

(ii) The requires-interface *GA* invokes the factory of module *RepairService* by executing the invoke activity $RepairService \bullet create!\langle OnRoadRepair, id_{ext}\rangle$. Hence, the following instance of *RepairService* is created:

$$
\begin{aligned}
&[id_{intra}]\,(\,ProvidesInt \mid RequiresInt \\
&\qquad \mid Wires \mid Components\,) \cdot \{x_{cust} \mapsto OnRoadRepair, x_{ext\_id} \mapsto id_{ext}\}
\end{aligned}
$$

GA also communicates the binding information to *CR* by invoking the operation *bindingInfo*.

(2) *OnRoadRepair initiates the conversation with RepairService.*

(i) The component *OR* of the *OnRoadRepair*'s instance executes transition *reqToGarage* corresponding to the interaction *bookGarage*🔔. The block **sends** of this transition corresponds to the COWS activity $OG_{roleA} \bullet bookGarage!\langle id_i, \text{🔔}, \text{"}gps = \ldots\text{"}\rangle$. Notably, in the implementation of component *OR* we take into account that it is connected to *GA* by means of the wire *OG*.

(ii) The wire *OG* catches the send event and adapts the endpoint of the activity of *OR* (i.e., *bookGarage*🔔) to the corresponding activity of the requires-interface *GA*. The executed COWS activity is $GA \bullet acceptBooking!\langle id_i, \text{🔔}, \text{"}gps = \ldots\text{"}\rangle$.

(iii) The requires-interface *GA* catches the message and replaces the identifier id_i inside the message with the external identifier id_{ext}. Then, it invokes operation *getRequest* provided by the module *RepairService*, i.e. it performs the COWS activity $RepairService \bullet getRequest!\langle id_{ext}, \text{🔔}, \text{"}gps = \ldots\text{"}\rangle$.

(iv) The message $\langle id_{ext}, \text{🔔}, \text{"}gps = \ldots\text{"}\rangle$ sent by *GA* is delivered to the instance of *RepairService* created at step (1-ii) by means of the correlation identifier id_{ext}. This instance can receive messages from the instance of *OnRoadRepair* through the provides-interface *CR*, that replaces the external identifier in the incoming messages with the internal identifier. Thus, $CG_{roleA} \bullet getRequest!\langle id_{intra}, \text{🔔}, \text{"}gps = \ldots\text{"}\rangle$ is executed.

(3) *RepairService processes the interaction and replies.*

(i) The implementation of the wire *CG* acts as that of *OG*, i.e. it just renames the endpoints according to its specification. Then, it catches the message $\langle id_{intra}, 🔔, \text{“}gps = \ldots\text{”}\rangle$ sent over the endpoint $CG_{roleA} \bullet getRequest$ and forwards it along $GO \bullet handleRequest$. Hence, the performed activity is $GO \bullet handleRequest!\langle id_{intra}, 🔔, \text{“}gps = \ldots\text{”}\rangle$. Notice that the component *GO* exploits the partner name *GO* to receive messages from other entities.

(ii) The implementation of *GO* executes transition *reqResp*. This means that it performs the activity $GO \bullet handleRequest?\langle id_{intra}, 🔔, x_d\rangle$ and replies with $CG_{roleB} \bullet handleRequest!\langle id_{intra}, ✉, \text{“Eur 75”}\rangle$, where “Eur 75” is the value returned by *computePrice*(“*gps* = . . . ”).

(iii) The wire *CG* catches the reply message, replaces the name of operation *handleRequest* with *getRequest* and forwards the message to *CR*. The executed activity is $CR \bullet getRequest!\langle id_{intra}, ✉, \text{“Eur 75”}\rangle$.

(iv) *CR* renames the operation *getRequest* in *acceptBookingResp*, replaces the internal identifier id_{intra} with the external one id_{ext}, and sends the reply message to the instance of module *OnRoadRepair*. The executed activity is $OnRoadRepair \bullet acceptBookingResp!\langle id_{ext}, ✉, \text{“Eur 75”}\rangle$. Notice that, if there were more than one instance of *OnRoadRepair*, the identifier id_{ext} would guarantee that the message is properly delivered to the (requires-interface of the) proper instance of *OnRoadRepair*.

(4) *OnRoadRepair receives and processes the reply.*

(i) *GA* catches the reply message, changes the operation name, replaces the identifier and forwards the message to *OG*. Thus, the executed activity is $OG_{roleB} \bullet acceptBooking!\langle id_i, ✉, \text{“Eur 75”}\rangle$.

(ii) *OG* changes again the name of the operation and delivers the message to the component *OR*. The executed activity is $OR \bullet bookGarage!\langle id_i, ✉, \text{“Eur 75”}\rangle$.

(iii) Finally, the receiving event triggers transition *respFromGarage* of *OR*, thus *OR*'s implementation executes $OR \bullet bookGarage?\langle id_i, ✉, x_{price}\rangle$.

It is worth noticing that, if during the above computation a fatal error occurs within the component *OR* of the *OnRoadRepair*'s instance under consideration (i.e., its instance state is set to *ERR*), the implementation of *OR* would execute a forced termination of the COWS term implementing *OR*. This is done by means of a kill activity **kill**(k).

6 Concluding Remarks

We presented some key aspects of the definition of an execution semantics for the modelling language SRML through an implementation in the process calculus COWS. Specifically, we aimed at providing a formal relationship between two different levels of abstraction that arise in SOC: the more declarative business modelling level that abstracts from the process of discovery, selection and binding available in the underlying SOA, and the more operational level where key aspects of service behaviour, including reconfiguration, message correlation and session management, need to be accounted for.

The architecture of the implementation was given a special emphasis. We consider this to be one of the main interests of our work in the sense that it reveals general aspects of what it means to implement a business modelling language over a calculus of services.

Indeed, our implementation is such that the structure of the COWS terms that implement SRML modules reflects the architecture of the configuration management process that is promoted through SRML. More precisely, we partition the implementation into areas of concern that derive from the declarative semantics of SRML [15], which has the advantage of permitting a modular and incremental development of the implementation.

So far, we have implemented the orchestration and the process through which reconfiguration takes place. These two aspects are not totally independent because the process of discovery and binding is triggered by events occurring in the execution of the components that orchestrate service execution. Therefore, our implementation takes into account the need for message correlation and the routing of messages to different instances of the same module or to different components with the same type in a module. In fact, the choice of using COWS to implement SRML, with respect to the many other calculi for SOC proposed in the literature (among which we want to mention [18,17,11,9,10,30]), has been mainly motivated by the need to easily support message correlation, together with implementation of shared states and forced termination of (parts of) services.

Related Work. Only a few attempts at providing a relationship between SOA languages set at different levels of abstraction have been proposed in the literature. In [24], UML4SOA, an UML-based domain-specific language, is used for modelling SOA artefacts, while WS-BPEL, Java and Jolie [1] are the target languages at operational level. While UML4SOA focuses on 'modelling service interactions, compensation, exception, and event handling', it does not abstract from the SOA middleware components in the same way as SRML e.g. discovery and selection need to be explicitly modelled. Another similar proposal is [29], which focuses on business process modelling and presents a translation of the Business Process Modeling Notation (BPMN) into the stochastic extension of COWS that enables quantitative reasoning by means of the probabilistic model checker PRISM. In [12], DecSerFlow and Event Calculus are used to specify constraints on the execution of service choreographies and, for verification purposes, both of them are mapped into SCIFF, a language introduced for specifying global interaction protocols, equipped with a proof procedure. Other work can be found in the literature where the focus is on executable languages such as WS-BPEL (for an overview see [27]). Many of these efforts aim at formalizing its semantics using Petri nets [27,22], but do not cover such dynamical aspects as service instantiation and message correlation. In general, anyway, WS-BPEL does not represent the architectural aspects of a service, which is instead one of the aims of SRML (which we recall is inspired by SCA).

Future Work. The implementation relies on specific properties of the middleware that COWS also abstracts from, in particular existence of a broker that performs service selection and of a constraint solver for SLAs. The refinement of the broker and of the constraint solver is a matter for future work, possibly based on existing work on dynamic and adaptive composition of autonomous services [1] and a dialect of COWS [21] that permits modelling QoS requirement specifications and SLA achievements. Such a refinement would provide a more detailed model of the process of matchmaking/ranking/selection, also based on SLAs, and of the process of negotiation. Another direction of further research concerns the use of the reasoning mechanisms and verification techniques that are being made available for COWS so that we can use particular properties of these processes of negotiation and

[1] http://www.jolie-lang.org/

matchmaking to reason about the dynamic aspects of SRML modules and configurations.

References

[1] The Dino Project, University College London. Web site: http://www.cs.ucl.ac.uk/research/dino/.

[2] Software engineering for service-oriented overlay computers (Sensoria). Web site: http://sensoria.fast.de/.

[3] J. Abreu and J.L. Fiadeiro. A coordination model for service-oriented interactions. In *COORDINATION*, volume 5052 of *LNCS*, pages 1–16. Springer, 2008.

[4] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer, 2004.

[5] J. Bauer, F. Nielson, H.R. Nielson, and H. Pilegaard. Relational Analysis of Correlation. In *SAS*, volume 5079 of *LNCS*, pages 32–46. Springer, 2008.

[6] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.

[7] L. Bocchi, J.L. Fiadeiro, A. Lapadula, R. Pugliese, and F. Tiezzi. From Architectural to Behavioural Specification of Services. Technical report, Università di Firenze, 2008. Available at http://rap.dsi.unifi.it/cows/.

[8] L. Bocchi, Y Hong, A Lopes, and J.L. Fiadeiro. From BPEL to SRML: A Formal Transformational Approach. In *Web Services and Formal Methods*, volume 4937 of *LNCS*, pages 92–107. Springer Verlag, 2007.

[9] M. Boreale, Roberto Bruni, Rocco De Nicola, and Michele Loreti. Sessions and pipelines for structured service programming. In G. Barthe and F.S. de Boer, editors, *FMOODS*, volume 5051 of *LNCS*, pages 19–38. Springer, 2008.

[10] R. Bruni, I. Lanese, H.C. Melgratti, and E. Tuosto. Multiparty Sessions in SOC. In *COORDINATION*, volume 5052 of *LNCS*, pages 67–82. Springer, 2008.

[11] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[12] F. Chesani, P. Mello, M. Montali, S. Storari, and P. Torroni. On the Integration of Declarative Choreographies and Commitment-based Agent Societies into the SCIFF Logic Programming Framework. *Journal of Multiagent and Grid Systems, Special Issue on Agents, Web Services and Ontologies: Integrated Methodologies*, 2009. To appear.

[13] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying COWS specifications. In *FASE*, volume 4961 of *LNCS*, pages 230–245. Springer, 2008.

[14] J.L. Fiadeiro, A. Lopes, and L. Bocchi. A formal approach to service component architecture. In *Web Services and Formal Methods*, volume 4184 of *LNCS*, pages 193–213. Springer, 2006.

[15] J.L. Fiadeiro, A. Lopes, and L. Bocchi. Semantics of service-oriented system configuration. Technical report, University of Leicester, 2008. Available at http://www.cs.le.ac.uk/people/jfiadeiro/.

[16] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[17] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: a calculus for service oriented computing. In *ICSOC*, volume 4294 of *LNCS*, pages 327–338. Springer, 2006.

[18] C. Laneve and G. Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.

[19] A. Lapadula, R. Pugliese, and F. Tiezzi. Regulating data exchange in service oriented applications. In *FSEN*, volume 4767 of *LNCS*, pages 223–239. Springer, 2007.

[20] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. Technical report, Università di Firenze, 2008. Available at http://rap.dsi.unifi.it/cows/. An extended abstract appeared in the proc. of ESOP'07.

[21] A. Lapadula, R. Pugliese, and F. Tiezzi. Service discovery and negotiation with COWS. In *WWV*, volume 200 of *ENTCS*, pages 133–154. Elsevier, 2008.

[22] N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. In *Web Services and Formal Methods*, volume 4937 of *LNCS*, pages 77–91. Springer, 2008.

[23] M. Beisiegel, H. Blohm, D. Booz, J. Dubray, A. Colyer, M. Edwards, D. Ferguson, B. Flood, M. Greenberg, D. Kearns, J. Marino, J. Mischkinsky, M. Nally, G. Pavlik, M. Rowley, K. Tam, and C. Trieloff. Building Systems using a Service Oriented Architecture. Whitepaper, SCA Consortium, 2005. http://www.oracle.com/technology/tech/webservices/standards/sca/pdf/SCA_White_Paper1_09.pdf.

[24] P. Mayer, A. Schroeder, and N. Koch. MDD4SOA: Model-Driven Service Orchestration. In *EDOC*, pages 203–212. IEEE Computer Society Press, 2008.

[25] R. Milner. *Communicating and Mobile Systems: The piCalculus*. Cambridge University Press, 1999.

[26] OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0, April 2007. Web site: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[27] C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, and H.M.W. Verbeek. Formal semantics and analysis of control flow in WS-BPEL (revised version). Technical report, BPM Center Report, 2005. http://www.BPMcenter.org.

[28] D. Prandi and P. Quaglia. Stochastic COWS. In *ICSOC*, volume 4749 of *LNCS*, pages 245–256. Springer, 2007.

[29] D. Prandi, P. Quaglia, and N. Zannone. Formal analysis of BPMN via a translation into COWS. In *COORDINATION*, volume 5052 of *LNCS*, pages 249–263. Springer, 2008.

[30] H.T. Vieira, L. Caires, and J. Costa Seco. The conversation calculus: A model of service-oriented computation. In Sophia Drossopoulou, editor, *ESOP2008*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.