From Architectural to Behavioural Specification of Services[⋆]

— An implementation of SRML into COWS —

L. Bocchi[1], J.L. Fiadeiro[1], A. Lapadula[2], R. Pugliese[2], and F. Tiezzi[2]

[1] Department of Computer Science, University of Leicester, UK
`{bocchi,jose}@mcs.le.ac.uk`

[2] Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, Italy
`{lapadula,pugliese,tiezzi}@dsi.unifi.it`

WORKING DRAFT – March 3, 2009

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.

Table of Contents

[⋆] This work is a revised and extended version of [8] and has been partially funded by the EU project Sensoria [2], IST-2005-016004.

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 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* [15]) 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* [22]) 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 [27]. 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, 16]) 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 [21], a stochastic extension to enable quantitative reasoning on service behaviours [29], 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 [14]. 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 a Backus-Naur Form syntax of SRML and introduces the case study that is used throughout the paper. Section 4 describes the architecture of the implementation through the case study. Section 5 reports the complete implementation of SRML into COWS. 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. Similarly, the overview of COWS gives only a glimpse of its semantics, a full account of which can be found in [22].

2.1 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* [6]) and is independent of the languages and platforms that are currently being provided for web [4] (or grid [17]) services. An encoding of WS-BPEL is available that illustrates how SRML (static) models can be (partially) implemented in more concrete languages [9].

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

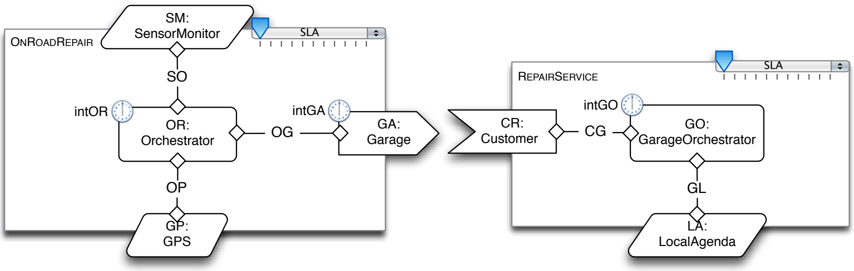

Fig. 1. Activity module OnRoadRepair (left-hand side) and service module RepairService (right-hand side)

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).

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

Table 1. COWS syntax

All interfaces involve a signature declaring the set of supported interactions and a specification of the behaviour associated with them. See [15] 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 [7].

2.2 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, y, ...). 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 [26]) 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 [22] 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 2.1 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.

```
(Module)
   M ::= MODULE mod is
           COMPONENTS COMPS
           PRVORSRV
           [REQUIRES REQS]
           [EXTERNAL POLICY SLAc]
           WIRES WIRES
           SPECIFICATIONS SPECS

(Provides or Serves Interface)
   PRVORSRV ::= PROVIDES pr : bp
              | SERVES pr : lp

(Components)
   COMPS ::= COMPS COMPS
           | comp : br
               [init IASGS]
               [term c]

(Initial assignments)
   IASGS ::= IASGS ∧ IASGS
           | lvar = e

(Requires interfaces)
   REQS ::= REQS REQS
          | req : bp [trigger c]

(Event type)
   ET ::= 🔔 | ✉ | ✓ | ✗ | ✞

(Wires)
   WIRES ::= WIRES WIRES
           | wire : name name
                      WLINES

(Wire lines)
   WLINES ::= WLINES WLINES
            | int ↔ int
                [: ET param ↔ param, ...
                 ..., ET param ↔ param]

(Specifications)
   SPECS ::= SPECS SPECS
           | BR | BP

(Business role)
   BR ::= BUSINESS ROLE br is
            INTERACTIONS INTS
            ORCHESTRATION
             [local LVARS]
             TRANS

(Business Protocol)
   BP ::= BUSINESS PROTOCOL bp is
            INTERACTIONS INTS
            BEHAVIOUR Description

(Layer Protocol)
   LP ::= LAYER PROTOCOL lp is
            INTERACTIONS INTS
            BEHAVIOUR Description
```

Table 2. SRML syntax (part 1/2)

In this section, we present a Backus-Naur Form syntax of SRML for those aspects that we considered in the implementation. Then, we use such syntax to textually describe the automotive scenario introduced in Section 2.1.

3.1 A Backus-Naur Form syntax of SRML

A Backus-Naur Form syntax of SRML is presented in Tables 2 and 3. The set of names is ranged over by *mod*, *pr*, *bp*, *lp*, *comp*, *br*, *req*, *wire*, *int*, *tr*, *param*, *type* and *lvar* used for a module, provides/serves-interface, business protocol, layer protocol, component, business role, requires-interface, wire, interaction, transition, parameter, type and local variable. The names of nodes in a SRML module, when we refer in general to either a provides-interface, a requires-interface or a component, are ranged over by *name*.

The language of *expressions*, ranged over by *e*, is deliberately omitted; we assume that expressions contain, at least, names and invocation of `ask` interactions, whose names differ from those of the expression functions (i.e. an expression cannot contain

(*Interactions*)
INTS ::= INTS INTS
| **rcv** *int* [🔔 PARAMS]
| **snd** *int* [🔔 PARAMS]
| **r&s** *int* [🔔 PARAMS
| [✉ PARAMS]]
| **s&r** *int* [🔔 PARAMS
| [✉ PARAMS]]
| **ask** *int*(TYPES) : *type*
| **rpl** *int*(TYPES) : *type*
| **tll** *int*(TYPES)
| **prf** *int*(TYPES)

(*Parameters*)
PARAMS ::= PARAMS , PARAMS
| *param* : *type*

(*Types*)
TYPES ::= TYPES , TYPES
| *type*

(*Local variables*)
LVARS ::= LVARS , LVARS
| *lvar* : *type*

(*Transitions*)
TRANS ::= TRANS TRANS
| **transition** *tr*
[**triggeredBy** TRIGS]
[**guardedBy** *c*]
[**effects** GASGS]
[**sends** GSENDS]

(*Trigger*)
TRIGS ::= *c* | *int* ET
| *int*($param_1, \ldots, param_n$)

(*Guarded assignments*)
GASGS ::= GASGS ∧ GASGS
| [*c* ⊃] ASG

(*Assignment*)
ASG ::= *lvar*[′] = *e*
| *int.param* = *e*

(*Guarded sends*)
GSENDS ::= GSENDS ∧ GSENDS
| [*c* ⊃] *int* [ET]
| [*c* ⊃] *int*($e_1, \ldots e_n$)
| [*c* ⊃] *int* ↰ [*e*]
| [*c* ⊃] ASG

Table 3. SRML syntax (part 2/2)

an invocation of the interaction **ask** *sqrt*(*integer*) : *integer* since *sqrt* is an expression function). A particular kind of expressions are the *conditions*, ranged over by *c*, whose evaluation is a boolean value. We will use $lvar'$ to denote the value that a state variable *lvar* has after the corresponding transition. The language is also parameterized by an unspecified set of *Service Level Agreement constraints* (ranged over by *SLAc*), and by an unspecified set of service descriptions (ranged over by Description) that represent the behavioural specifications of abstract references (i.e., provides-/serves-/requires-interfaces in SRML).

The syntax of the module definition is given in Table 2. A module M is defined by a number of components COMPS, one provides-interface/serves-interface, a number of requires-interfaces REQS, one external policy *SLAc*, a number of wires and specifications SPECS. We do not include uses-interfaces because we did not defined their implementation in COWS yet. COMPS represents a set of one or more components, each defined by a name *comp* and a type *br* that refers to one BR element in the specifications SPECS, and equipped with a set of initial assignments and a termination condition. An external provides-interface is defined by a name *pr* and a type *bp* that refers to one BP element in SPECS. A serves-interface is defined by a name *pr* and a type *lp* that refers to one LP element in SPECS. REQS represents a set of one or more requires-interfaces, each defined by a name *req* and a type *bp* that refers to one BP element in the specifications SPECS. Each external interface is equipped with a trigger condition that launches the discovery. SPECS is the set of specifications, which can be business roles (BR), business protocol BP and layer protocol (LP) elements.

The syntax of the specifications referred to by a module definition is given in Table 3. The business role BR is defined by one declaration of interactions (i.e. a syntactical interface) INTS and one orchestration description. INTS represents the interactions supported by SRML. There are different types of interaction: asynchronous one-way (i.e., receive **rcv** or send **snd**), asynchronous conversational (i.e., receive-and-send **r&s**, or send-and-receive **s&r**), and synchronous (i.e. ask **ask**, reply **rpl**, tell **tll**, and perform **prf**). A number of interaction events is associated with each conversational interaction: an initiation event (denoted by 🔔), a reply-event (denoted by ⊠), a commit-event (denoted by ✓), a cancel-event (denoted by 8), and a revoke-event (denoted by ✞). Interactions can involve a number of parameters for each phase of the conversation (i.e., 🔔-parameters for the initiation and ⊠-parameters for the reply). One-way interactions have associated only one 🔔-event.

The orchestration consists of an optional declaration of local variables LVARS and one or more transitions TRANS. A transition has (1) an optional trigger TRIGS that is either a condition, a receive event or a receive of a synchronous interaction (when a trigger is not specified we consider the default condition to be true), (2) an optional guard that is a condition (where true is the default condition), (3) optional effects (i.e., a number of assignments GASGS), and (4) an optional sends section represented by the term GSENDS consisting in one or more send interaction events, sends of synchronous interactions, return events for **rpl** and **prf** interactions (denoted by *int* ↰ [*e*]), and assignments to output parameters. The interaction events and assignments in GSENDS may have a condition, likewise assignments in GASGS.

3.2 SRML textual specification of the automotive case study

Table 4 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*). Recall that uses-interfaces are not considered here since 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.

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,

```
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 4. The textual definition of the module *OnRoadRepair*

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.

```
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 5. The textual definition of the module *RepairService*

The transition sends the event *bookGarage*🔔 and assigns the sensor data (stored in the local variable *data*) to the parameter *bookGarage.data*. 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 5. 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 engaged 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.

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 dis-

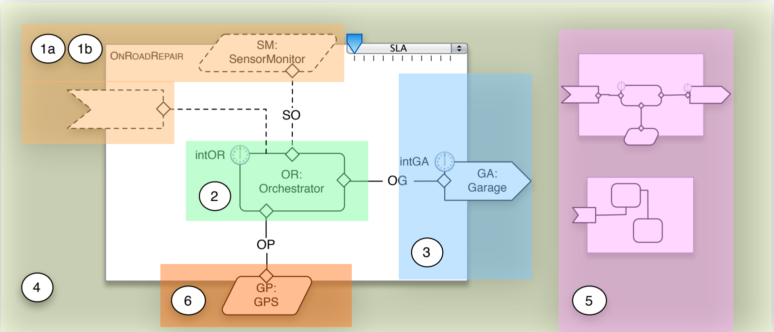

Fig. 2. Decomposition of OnRoadRepair into areas of concern

covery 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. 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.

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 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.

4.1 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

$$\begin{array}{l}\langle\!\langle \mathtt{MODULE}\ OnRoadRepair\ \mathtt{is}\ \ldots\rangle\!\rangle \mid \langle\!\langle \mathtt{MODULE}\ RepairService\ \mathtt{is}\ \ldots\rangle\!\rangle\\ \mid Middleware \mid Environment \mid BottomLayer\end{array}$$

where *Middleware* is the term ($Broker \mid Registry \mid ConstraintSolver \mid MatchmakingAgent \mid \ldots$), 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:

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

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:

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

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* • *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* • *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:

$$\begin{array}{l}
*\,[x_{data}]\, OG_{roleA} \bullet bookGarage?\langle id_i, \Bell, x_{data}\rangle.\, GA \bullet acceptBooking!\langle id_i, \Bell, x_{data}\rangle \\
\mid\ *\,[x_{servicePrice}]\, OG_{roleB} \bullet acceptBooking?\langle id_i, \Letter, x_{servicePrice}\rangle. \\
\qquad\qquad\qquad OR \bullet bookGarage!\langle id_i, \Letter, x_{servicePrice}\rangle
\end{array}$$

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 4). 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:

$$
\begin{array}{l}
GA \bullet trigger?\langle id_i \rangle. \\
(\, Broker \bullet disc!\langle OnRoadRepair, id_i, \text{``}Garage\ \mathtt{\textbf{is}}\ \ldots\text{''}, carUserSLAconstraints \rangle \\
\ \mid [x_p, x_{acceptBooking}, x_{binding}]\, OnRoadRepair \bullet GA?\langle id_i, x_p, x_{acceptBooking}, x_{binding} \rangle. \\
\qquad [id_{ext}]\, (\, x_p \bullet create!\langle OnRoadRepair, id_{ext} \rangle \\
\qquad\qquad\qquad \mid x_p \bullet bindingInfo!\langle id_{ext}, x_{binding} \rangle \\
\qquad\qquad\qquad \mid *\, [x_{info}]\, GA \bullet acceptBooking?\langle id_i, \text{🔔}, x_{info} \rangle. \\
\qquad\qquad\qquad\quad (\, x_p \bullet x_{acceptBooking}!\langle id_{ext}, \text{🔔}, x_{info} \rangle \\
\qquad\qquad\qquad\qquad \mid [x_{servicePrice}]\, OnRoadRepair \bullet acceptBooking?\langle id_{ext}, \text{✉}, x_{servicePrice} \rangle. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad OG_{roleB} \bullet acceptBooking!\langle id_i, \text{✉}, x_{servicePrice} \rangle\,) \\
\qquad\qquad\qquad \mid \ldots\,)\,)
\end{array}
$$

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* • *trigger*, which is sent by the implementation of the component *OR* when the instance state is set to *READY* by transition *data_receiving*. Notably, in this case the binding information (stored in $x_{binding}$) is the operation name *acceptBooking* used by the implementation of *GA* to receive the response related to the corresponding **`r&s`** interaction.

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

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

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 *acceptBooking*).

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.

4.2 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.*

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

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

2. 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:

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

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

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

1. 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*.
2. 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 $GA \bullet acceptBooking!\langle id_i, \text{🔔}, \text{“}gps = \ldots\text{”}\rangle$.
3. 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 $RepairService \bullet getRequest!\langle id_{ext}, \text{🔔}, \text{“}gps = \ldots\text{”}\rangle$.
4. 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.*

1. 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 $\langle id_{intra}, \text{🔔}, \text{“}gps = \ldots\text{”}\rangle$ sent over the endpoint $CG_{roleA} \bullet getRequest$ and forwards it along $GO \bullet handleRequest$. Hence, the performed activity is $GO \bullet handleRequest!\langle id_{intra}, \text{🔔}, \text{“}gps = \ldots\text{”}\rangle$. Notice that the component *GO* exploits the partner name *GO* to receive messages from other entities.
2. The implementation of *GO* executes transition *reqResp*. This means that it performs the activity $GO \bullet handleRequest?\langle id_{intra}, \text{🔔}, x_d\rangle$ and replies with $CG_{roleB} \bullet handleRequest!\langle id_{intra}, \text{✉}, \text{“Eur 75”}\rangle$, where “Eur 75” is the value returned by $computePrice(\text{“}gps = \ldots\text{”})$.
3. 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 $CR \bullet getRequest!\langle id_{intra}, \text{✉}, \text{“Eur 75”}\rangle$.
4. *CR* renames the operation *getRequest* in *acceptBooking*, 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 $OnRoadRepair \bullet acceptBooking!\langle id_{ext}, \text{✉}, \text{“Eur 75”}\rangle$. 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.*

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

2. OG changes again the name of the operation and delivers the message to the component OR. The executed activity is $OR \bullet bookGarage!\langle id_i, \boxtimes, \text{“Eur 75”}\rangle$.
3. Finally, the receiving event triggers transition *respFromGarage* of OR, thus OR’s implementation executes $OR \bullet bookGarage?\langle id_i, \boxtimes, x_{price}\rangle$.

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 $\mathbf{kill}(k)$.

5 Implementing SRML into COWS

In this section, we report the complete implementation of SRML into COWS. The presentation has been split in two parts: firstly we discuss the implementation of SRML modules, then the implementation of business roles. In the sequel, we will use $\hat{n}$ to stand for the endpoint $n_p \bullet n_o$ or for the tuple $\langle n_p, n_o\rangle$ and rely on the context to resolve any ambiguity.

5.1 Implementation of modules

We present here the implementation $\langle\!\langle\cdot\rangle\!\rangle$ of SRML modules into COWS terms. The function $\langle\!\langle\cdot\rangle\!\rangle$ is defined by induction on the syntax of modules as follows [3].

$$
\begin{array}{lcl}
\langle\!\langle \texttt{MODULE}\ mod\ \texttt{is} & & Broker \bullet pub!\langle mod, \llbracket \mathrm{SPECS} \rrbracket^{\mathcal{P}}_{bp}, SLAc\rangle \\
\quad \texttt{COMPONENTS}\ \mathrm{COMPS} & & \mid\ *\, [x_{cust}, x_{ext\_id}]\ mod \bullet create?\langle x_{cust}, x_{ext\_id}\rangle. \\
\quad \texttt{PROVIDES}\ pr\ :\ bp & & \quad [id_{intra}]\ (\ \langle\!\langle \mathrm{SPECS} \rangle\!\rangle_{(mod,pr,bp,\llbracket \mathrm{WIRES} \rrbracket^{\mathcal{W}}_{pr})} \\
\quad \texttt{REQUIRES}\ \mathrm{REQS} & = & \qquad\qquad \mid \langle\!\langle \mathrm{SPECS} \rangle\!\rangle_{(mod,SLAc,\llbracket \mathrm{REQS},\mathrm{WIRES} \rrbracket^{\mathcal{R}})} \\
\quad \texttt{EXTERNAL POLICY}\ SLAc & & \qquad\qquad \mid \langle\!\langle \mathrm{SPECS} \rangle\!\rangle_{(\mathrm{WIRES},\llbracket \mathrm{COMPS} \rrbracket^{\mathcal{C}})} \\
\quad \texttt{WIRES}\ \mathrm{WIRES} & & \qquad\qquad \mid \langle\!\langle \mathrm{WIRES} \rangle\!\rangle\ ) \\
\quad \texttt{SPECIFICATIONS}\ \mathrm{SPECS}\rangle\!\rangle & &
\end{array}
$$

where

- function $\llbracket \mathrm{SPECS} \rrbracket^{\mathcal{P}}_{bp}$ returns the specification of the business role bp defined in SPECS;
- function $\llbracket \mathrm{WIRES} \rrbracket^{\mathcal{W}}_{pr}$ returns the set W of pairs of the form $(wire, int)$ such that *wire* is a wire in WIRES connected to the provides-interface *pr* involving the interaction *int*;
- function $\llbracket \mathrm{REQS}, \mathrm{WIRES} \rrbracket^{\mathcal{R}}$ returns the set R of quadruples of the form $(req, bp, int, wire)$ such that $req\ :\ bp$ is a definition in REQS and *wire* is a wire in WIRES connected to *req* and involving the interaction *int*;

---

[3] For each SRML term having optional elements, we only show the implementation for the full case. From this, the remaining cases can be trivially obtained.

- function $[\![\mathrm{COMPS}]\!]^{C}$ returns the set C of quadruples of the form $(comp, br, \mathrm{IASGS}, c)$ such that $comp : br$ **init** IASGS **term** c is a definition in COMPS.

The implementation $\langle\!\langle\cdot\rangle\!\rangle$ relies on the three auxiliary (parametric) functions $\langle\!\langle\cdot\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})}$, $\langle\!\langle\cdot\rangle\!\rangle_{(mod,SLAc,\mathsf{R})}$, and $\langle\!\langle\cdot\rangle\!\rangle_{(\mathrm{WIRES},\mathsf{C})}$, for the provides-interface, the requires-interfaces, and the components, respectively. Auxiliary functions $[\![\cdot]\!]^{\mathcal{P}}_{\cdot}$, $[\![\cdot]\!]^{\mathcal{W}}_{\cdot}$, $[\![\cdot]\!]^{\mathcal{R}}$ and $[\![\cdot]\!]^{C}$ can be inductively defined; however, since their definitions are straightforward, to save space, they are not shown here.

For the sake of simplicity, the implementation is given only for (the more general) service modules. Indeed, the implementation of activity modules is (almost) the same of that of service modules. As highlighted in the example presented in Section 4, the difference between service and activity modules affects mainly the COWS representations of the invoking entities.

Provides-interfaces. The implementation of provides-interfaces $\langle\!\langle\cdot\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})}$ is defined as follows.

- $\langle\!\langle \mathrm{SPECS}_1\ \ \mathrm{SPECS}_2\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \ = \ \langle\!\langle \mathrm{SPECS}_1\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \mid \langle\!\langle \mathrm{SPECS}_2\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})}$
- $\langle\!\langle \mathrm{BR}\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \ = \ \mathbf{0}$
- $\langle\!\langle \texttt{BUSINESS PROTOCOL}\ bp'\ \texttt{is}\ \ldots\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \ = \ \mathbf{0} \qquad \text{if } bp' \neq bp$
- $\langle\!\langle \texttt{BUSINESS PROTOCOL}\ bp\ \texttt{is}$
  $\texttt{INTERACTIONS}\ \mathrm{INTS}$
  $\texttt{BEHAVIOUR}\ \mathrm{Description}\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \ =$
  $[\mathsf{Var}(\mathrm{INTS})]\, mod \bullet bindingInfo?\langle x_{ext\_id}, \mathsf{Var}(\mathrm{INTS})\rangle.\ \langle\!\langle \mathrm{INTS}\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})}$

  where the auxiliary function $\mathsf{Var}(\mathrm{INTS})$ returns an ordered list of COWS variables x_{int} such that int is an interaction of type **s&r**, **r&s**, **rcv** or **prf** defined in INTS. Its inductive definition is straightforward.
- $\langle\!\langle \mathrm{INTS}_1\ \ \mathrm{INTS}_2\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \ = \ \langle\!\langle \mathrm{INTS}_1\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \mid \langle\!\langle \mathrm{INTS}_2\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})}$
- $\langle\!\langle \texttt{snd}\ int \Bell\ param_1 : type_1, \ldots, param_n : type_n\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \ =$
  $*\,[x_{param_1}, \ldots, x_{param_n}]\, mod \bullet int?\langle x_{ext\_id}, \Bell, x_{param_1}, \ldots, x_{param_n}\rangle.$
  $[\hat{n}]\,(\,\hat{n}!\langle\rangle \mid \sum_{(wire,int)\in\mathsf{W}} \hat{n}?\langle\rangle.\ wire_{roleA} \bullet int!\langle id_{intra}, \Bell, x_{param_1}, \ldots, x_{param_n}\rangle\,)$
- $\langle\!\langle \texttt{rcv}\ int \Bell\ param_1 : type_1, \ldots, param_n : type_n\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \ =$
  $*\,[x_{param_1}, \ldots, x_{param_n}]\, pr \bullet int?\langle id_{intra}, \Bell, x_{param_1}, \ldots, x_{param_n}\rangle.$
  $x_{cust} \bullet x_{int}!\langle x_{ext\_id}, \Bell, x_{param_1}, \ldots, x_{param_n}\rangle$
- $\langle\!\langle \texttt{s\&r}\ int \Bell\ param_1 : type_1, \ldots, param_n : type_n$
  $\Letter\ param'_1 : type'_1, \ldots, param'_m : type'_m\rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \ =$
  $*\,[x_{param_1}, \ldots, x_{param_n}]\, mod \bullet int?\langle x_{ext\_id}, \Bell, x_{param_1}, \ldots, x_{param_n}\rangle.$
  $(\,[\hat{n}]\,(\,\hat{n}!\langle\rangle \mid \sum_{(wire,int)\in\mathsf{W}} \hat{n}?\langle\rangle.\ wire_{roleA} \bullet int!\langle id_{intra}, \Bell, x_{param_1}, \ldots, x_{param_n}\rangle\,)$
  $\mid\ [x_{param'_1}, \ldots, x_{param'_m}]\, pr \bullet int?\langle id_{intra}, \Letter, x_{param'_1}, \ldots, x_{param'_m}\rangle.$
  $x_{cust} \bullet x_{int}!\langle x_{ext\_id}, \Letter, x_{param'_1}, \ldots, x_{param'_m}\rangle\,)$

- $\langle\!\langle \textbf{\texttt{r\&s}}\ int\ \Bell\ param_1 : type_1, \ldots, param_n : type_n$
  $\qquad\qquad \Letter\ param'_1 : type'_1, \ldots, param'_m : type'_m \rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \ = $
  $* \, [x_{param_1}, \ldots, x_{param_n}] \, pr \bullet int?\langle id_{intra}, \Bell, x_{param_1}, \ldots, x_{param_n} \rangle.$
  $\quad (\, x_{cust} \bullet x_{int}!\langle x_{ext\_id}, \Bell, x_{param_1}, \ldots, x_{param_n} \rangle$
  $\quad\ \mid \ [x_{param'_1}, \ldots, x_{param'_m}] \, mod \bullet int?\langle x_{ext\_id}, \Letter, x_{param'_1}, \ldots, x_{param'_m} \rangle.$
  $\qquad [\hat{n}] \, (\, \hat{n}!\langle\rangle \mid \sum_{(wire,int)\in \mathsf{W}} \hat{n}?\langle\rangle.\, wire_{roleA} \bullet int!\langle id_{intra}, \Letter, x_{param'_1}, \ldots, x_{param'_m} \rangle \,)\,)$

- $\langle\!\langle \textbf{\texttt{rpl}}\ int(type_1, \ldots, type_n) : type \rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \ = $
  $* \, [x_1, \ldots, x_n, x_r] \, pr \bullet int?\langle id_{intra}, x_1, \ldots, x_n, x_r \rangle.\, x_{cust} \bullet int!\langle x_{ext\_id}, x_1, \ldots, x_n, x_r \rangle$

- $\langle\!\langle \textbf{\texttt{ask}}\ int(type_1, \ldots, type_n) : type \rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \ = $
  $* \, [x_1, \ldots, x_n, x_r] \, mod \bullet int?\langle x_{ext\_id}, x_1, \ldots, x_n, x_r \rangle.$
  $\quad [\hat{n}] \, (\, \hat{n}!\langle\rangle \mid \sum_{(wire,int)\in \mathsf{W}} \hat{n}?\langle\rangle.\, wire_{roleA} \bullet int!\langle id_{intra}, , x_1, \ldots, x_n, x_r \rangle \,)\,)$

- $\langle\!\langle \textbf{\texttt{prf}}\ int(type_1, \ldots, type_n) \rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \ = $
  $* \, [x_1, \ldots, x_n] \, pr \bullet int?\langle id_{intra}, x_1, \ldots, x_n \rangle.\, x_{cust} \bullet x_{int}!\langle x_{ext\_id}, x_1, \ldots, x_n \rangle$

- $\langle\!\langle \textbf{\texttt{tll}}\ int(type_1, \ldots, type_n) \rangle\!\rangle_{(mod,pr,bp,\mathsf{W})} \ = $
  $* \, [x_1, \ldots, x_n] \, mod \bullet int?\langle x_{ext\_id}, x_1, \ldots, x_n \rangle.$
  $\quad [\hat{n}] \, (\, \hat{n}!\langle\rangle \mid \sum_{(wire,int)\in \mathsf{W}} \hat{n}?\langle\rangle.\, wire_{roleA} \bullet int!\langle id_{intra}, , x_1, \ldots, x_n \rangle \,)\,)$

The function $\langle\!\langle \cdot \rangle\!\rangle_{(mod,pr,bp,\mathsf{W})}$ returns the COWS empty term **0** if the argument is either a business role or a business protocol different from the parameter *bp*. The implementation of the business protocol *bp* is the parallel composition of the implementations of its interaction declarations preceded by a receive activity for collecting binding information. In particular, the implementation of a **`snd`** interaction declaration is a persistent COWS term that receives messages from the external environment and forwards them to one of the wires connected to the provides-interface. Conversely, the implementation of a **`rcv`** interaction catches messages generated by a component of the module (and transmitted by a wire) and sends them to an (instance of) another module. The implementation of the interaction declarations of type **`s&r`** and **`r&s`** are a mix of the previous ones. Implementations of synchronous interactions are similar: implementations of **`rpl`** and **`ask`** behave as the implementation of **`rcv`** and **`snd`**, respectively, while implementations of **`prf`** and **`tll`** are obtained from those for **`rpl`** and **`ask`** by removing the variable x_r storing the endpoint for the callbacks. Recall that the types of interactions of a business protocols associated to a provides-interface are defined from the point of view of the invoker (see, e.g., the interaction *getRequest* of type **`s&r`** in Table 5). Notice also that communication along the endpoint $\hat{n}$ is used to deal with the case when an interaction of the interface is connected to more than one component at the same time by means of different wires. In this case, incoming messages are non-deterministically routed to the wires.

Requires-interfaces. The implementation of requires-interfaces $\langle\!\langle \cdot \rangle\!\rangle_{(mod,SLAc,\mathsf{R})}$ is defined as follows.

- $\langle\!\langle \mathrm{SPECS}_1 \ \ \mathrm{SPECS}_2 \rangle\!\rangle_{(mod,SLAc,\mathsf{R})} \ = \ \langle\!\langle \mathrm{SPECS}_1 \rangle\!\rangle_{(mod,SLAc,\mathsf{R})} \ \mid \ \langle\!\langle \mathrm{SPECS}_2 \rangle\!\rangle_{(mod,SLAc,\mathsf{R})}$

- $\langle\!\langle \mathrm{BR} \rangle\!\rangle_{(mod,SLAc,\mathsf{R})} \;=\; \mathbf{0}$

- if $\forall\,(req, bp, int, wire) \in \mathsf{R}\,.\,bp \neq bp'$
  $\langle\!\langle \mathtt{BUSINESS\ PROTOCOL}\ bp'\ \mathtt{is}\ \ldots \rangle\!\rangle_{(mod,SLAc,\mathsf{R})} \;=\; \mathbf{0}$

- $$\begin{array}{l}
\langle\!\langle \mathtt{BUSINESS\ PROTOCOL}\ bp\ \mathtt{is} \\
\quad \mathtt{INTERACTIONS}\ \mathrm{INTS} \\
\quad \mathtt{BEHAVIOUR}\ \mathrm{Description} \rangle\!\rangle_{(mod,SLAc,\mathsf{R})} \;=\; \\
\quad \prod_{(req,bp,int,wire)\in\mathsf{R}}\ req \bullet trigger?\langle id_{intra}\rangle. \\
\qquad (\, Broker \bullet disc!\langle mod, id_{intra}, bp, SLAc\rangle \\
\qquad\quad \mid [x_p, \mathsf{Var}'(\mathrm{INTS}), \bar{x}_{binding}]\, mod \bullet req?\langle id_{intra}, x_p, \mathsf{Var}'(\mathrm{INTS}), \bar{x}_{binding}\rangle. \\
\qquad\qquad [id_{ext}]\, (\, x_p \bullet create!\langle mod, id_{ext}\rangle \mid x_p \bullet bindingInfo!\langle id_{ext}, \bar{x}_{binding}\rangle \\
\qquad\qquad\qquad \mid \langle\!\langle \mathrm{INTS} \rangle\!\rangle_{(mod,req,bp,int,wire)}\ )\ )
\end{array}$$

  where the auxiliary function $\mathsf{Var}'(\mathrm{INTS})$ returns an ordered list of COWS variables x_{int} such that *int* is an interaction of type **r&s**, **s&r**, **snd**, **ask** or **tll** defined in INTS. Its inductive definition is straightforward.

- $$\begin{array}{l}
\langle\!\langle \mathrm{INTS}_1\ \ \mathrm{INTS}_2 \rangle\!\rangle_{(mod,req,bp,int,wire)} \;=\; \\
\quad \langle\!\langle \mathrm{INTS}_1 \rangle\!\rangle_{(mod,req,bp,int,wire)} \mid \langle\!\langle \mathrm{INTS}_2 \rangle\!\rangle_{(mod,req,bp,int,wire)}
\end{array}$$

- $\langle\!\langle \mathtt{r\&s}\ int'\ \ldots \rangle\!\rangle_{(mod,req,bp,int,wire)} \;=\; \mathbf{0} \qquad \text{if } int \neq int'$

- $$\begin{array}{l}
\langle\!\langle \mathtt{r\&s}\ int\ \Bell\ param_1 : type_1, \ldots, param_n : type_n \\
\qquad\qquad \Letter\ param'_1 : type'_1, \ldots, param'_m : type'_m \rangle\!\rangle_{(mod,req,bp,int,wire)} \;=\; \\
\quad *\,[x_{param_1}, \ldots, x_{param_n}]\, req \bullet int?\langle id_{intra}, \Bell, x_{param_1}, \ldots, x_{param_n}\rangle. \\
\qquad (\, x_p \bullet x_{int}!\langle x_{ext\_id}, \Bell, x_{param_1}, \ldots, x_{param_n}\rangle \\
\qquad\quad \mid\ [x_{param'_1}, \ldots, x_{param'_m}]\, mod \bullet int?\langle x_{ext\_id}, \Letter, x_{param'_1}, \ldots, x_{param'_m}\rangle. \\
\qquad\qquad\qquad\qquad\qquad wire_{roleB} \bullet int!\langle id_{intra}, \Letter, x_{param'_1}, \ldots, x_{param'_m}\rangle\ )
\end{array}$$

Similarly to the implementation of provides-interfaces, the function $\langle\!\langle \cdot \rangle\!\rangle_{(mod,SLAc,\mathsf{R})}$ returns the COWS empty term $\mathbf{0}$ if the argument is either a business role or a business protocol different from those belonging to R. A requires-interface is implemented in a COWS term that waits for a triggering message along the endpoint $req \bullet trigger$ by a component connected to the interface. The term cannot evaluate the triggering condition by itself, because the variables contained in the condition are local to the component. Notably, for the sake of simplicity, we assume here that a triggering condition depends on the internal state of only one component. After that the trigger is fired, a discovery process starts: the term sends a message with the business protocol and the external policy to *Broker* and waits for a message with the binding information (i.e. two ordered lists of operation names). Then, a new instance of the discovered module is created, the binding information (regarding to the operation used to receive messages from the invoked service) are sent to that instance, and from this moment the COWS term can properly handle the interactions with the instance. The implementation of the interactions declared in the requires-interfaces is symmetric to that of the interactions of the provides-interfaces. We show as example the implementation of an interaction of type **r&s**. Notably, the binding information are exploited to guarantee a correct communication between the two modules (see the use of x_{int} and int in the implementation of the **`r&s`** interaction).

Components. The implementation of components $\langle\!\langle\cdot\rangle\!\rangle_{(\mathrm{WIRES},\mathsf{C})}$ is defined as follows.

- $\langle\!\langle \mathrm{SPECS}_1 \;\; \mathrm{SPECS}_2 \rangle\!\rangle_{(\mathrm{WIRES},\mathsf{C})} \;=\; \langle\!\langle \mathrm{SPECS}_1 \rangle\!\rangle_{(\mathrm{WIRES},\mathsf{C})} \;|\; \langle\!\langle \mathrm{SPECS}_2 \rangle\!\rangle_{(\mathrm{WIRES},\mathsf{C})}$

- $\langle\!\langle \mathrm{BP} \rangle\!\rangle_{(\mathrm{WIRES},\mathsf{C})} \;=\; \mathbf{0}$

- if $\forall\,(comp, br, \mathrm{IASGS}, c) \in \mathsf{C} \;.\; br \neq br'$
  $\langle\!\langle \textbf{\texttt{BUSINESS ROLE}}\ br'\ \textbf{\texttt{is}}\ \ldots \rangle\!\rangle_{(\mathrm{WIRES},\mathsf{C})} \;=\; \mathbf{0}$

- $\langle\!\langle \textbf{\texttt{BUSINESS ROLE}}\ br\ \textbf{\texttt{is}}$
  $\quad \textbf{\texttt{INTERACTIONS}}\ \mathrm{INTS}$
  $\quad \textbf{\texttt{ORCHESTRATION}}\ \textbf{\texttt{local}}\ \mathrm{LVARS}$
  $\qquad\qquad\qquad \mathrm{TRANS} \rangle\!\rangle_{(\mathrm{WIRES},\mathsf{C})} \;=$
  $\prod_{(comp,br,\mathrm{IASGS},c)\in\mathsf{C}} [\hat{n}, k_{term}, \mathsf{varName}(\mathrm{LVARS}), \mathsf{paramName}(\mathrm{INTS}), set, get]$
  $(\,\langle\!\langle \mathrm{LVARS} \rangle\!\rangle \;|\; \langle\!\langle \mathrm{INTS} \rangle\!\rangle \;|\; \langle\!\langle \mathrm{IASGS} \rangle\!\rangle_{\hat{n}} \;|\; Scheduler_c \;|\; ReqTriggerEval_{[\![comp,\mathrm{WIRES}]\!]}$
  $\;|\; \hat{n}?\langle\rangle.\, \langle\!\langle \mathrm{TRANS} \rangle\!\rangle^{[\![\mathrm{INTS}]\!]}_{comp} \cdot \mathsf{R}(\mathrm{INTS}, \mathrm{WIRES}, comp)\,)$

  where
  - set and get are (restricted) operation names used by the implementation of transitions;
  - function $\mathsf{varName}(\mathrm{LVARS})$ returns the list of variable names in LVARS;
  - function $\mathsf{paramName}(\mathrm{INTS})$ returns the list of names of the form $int.param$ such that $param$ is a parameter of an interaction int in INTS;
  - function $[\![comp, \mathrm{WIRES}]\!]$ returns the set Cond of pairs of the form (req, c) such that req is a requires-interface connected to the component $comp$ by a wire in WIRES;
  - function $[\![\mathrm{INTS}]\!]$ returns the interaction environment $\mathcal{E}$ storing information about types and parameters of interactions (see the next section for more details).
  - function $\mathsf{R}(\mathrm{INTS}, \mathrm{WIRES}, comp)$ returns a renaming from interaction names to wires names (with role); in particular, an interaction int is renamed in $wire_{roleA}$ (resp. $wire_{roleB}$) if int is an interaction in INTS and $wire$ is a wire such that in WIRES there exists the declaration $wire \;:\; comp \;\; name \;\; int \leftrightarrow int' \ldots$ (resp. $wire \;:\; name \;\; comp \;\; int' \leftrightarrow int \ldots$);
  - $Scheduler_c$ is the COWS term $(check\&go_c \;|\; *\,\widehat{end}?\langle\rangle.\, check\&go_c)$ where

    $$check\&go_c \triangleq [\hat{n}, \hat{m}]\,(\,\langle\!\langle c \rangle\!\rangle_{\hat{m},\hat{n}} \mid \hat{m}?\langle \mathsf{true}, \hat{n}\rangle.\, \mathbf{kill}(k_{term}) \;+\; \hat{m}?\langle \mathsf{false}, \hat{n}\rangle.\, \widehat{start}!\langle\rangle\,)$$

  - $ReqTriggerEval_{\mathsf{Cond}}$ is the COWS term

    $$\begin{aligned}&\textstyle\prod_{(req,c)\in\mathsf{Cond}} [\hat{l}](\,\hat{l}!\langle\rangle\\ &\qquad \mid *\,\hat{l}?\langle\rangle.\, \widehat{start}?\langle\rangle.\, [\hat{n}, \hat{m}]\,(\,\langle\!\langle c \rangle\!\rangle_{\hat{m},\hat{n}} \mid \hat{m}?\langle \mathsf{true}, \hat{n}\rangle.\, (req \bullet trigger!\langle id_{intra}\rangle \mid \widehat{end}!\langle\rangle)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad + \hat{m}?\langle \mathsf{false}, \hat{n}\rangle.\, (\hat{l}!\langle\rangle \mid \widehat{end}!\langle\rangle)\,)\,)\end{aligned}$$

The above functions can easily inductively defined.

The implementation of a component is the parallel composition of the implementations of its local variables, interaction declarations, initial assignments, and transitions (these implementations are presented in the next section). Other two parallel term are as follows: $\mathit{Scheduler}_c$ and $\mathit{ReqTriggerEval}_{\mathsf{Cond}}$. The former one guarantees the atomic execution of the transition, i.e. when a transition is triggered by an orchestration, other transitions on such orchestration are suspended until the first transition has completed. To trigger a transition, the orchestration acquires the shared lock $\widehat{start}$, that is reinstalled by a signal along the endpoint $\widehat{end}$ when the transition has finished. Each time, the termination condition of the component is checked and, in case it holds, the execution of the component is immediately stopped by invoking the forced termination. The modelled scheduler chooses the transition to execute non-deterministically, so we rely on a *fairness assumption* to guarantee progress properties: if an implementation of a transition can acquire the lock, then eventually this synchronization will succeed. Of course, more complex schedulers (that, e.g., avoid fairness assumptions) could be defined. The term $\mathit{ReqTriggerEval}_{\mathsf{Cond}}$ checks the triggering conditions of the requires-interfaces connected to the component repeatedly. At the beginning, the initial assignments are performed and, then, their termination triggers the execution of the transitions. Notably, the initial delimitation activity permits localizing the scope of the variables and interaction parameters.

Wires. The implementation of a wire is in some way similar to those of interfaces. Indeed, it is a COWS term that catches messages along an endpoint and simply forwards them along another endpoint according to the nodes connected by the wire and the interactions involved.

- $\langle\!\langle \text{WIRES}\ \ \text{WIRES} \rangle\!\rangle \;=\; \langle\!\langle \text{WIRES} \rangle\!\rangle \mid \langle\!\langle \text{WIRES} \rangle\!\rangle$

- $\langle\!\langle \mathit{wire} \;:\; \mathit{name}_1\ \ \mathit{name}_2\ \ \ \text{WLINES} \rangle\!\rangle \;=\; \langle\!\langle \text{WLINES} \rangle\!\rangle_{(\mathit{wire},\mathit{name}_1,\mathit{name}_2)}$

- $\langle\!\langle \text{WLINES}\ \ \text{WLINES} \rangle\!\rangle_{(\mathit{wire},\mathit{name}_1,\mathit{name}_2)} \;=$
  $\langle\!\langle \text{WLINES} \rangle\!\rangle_{(\mathit{wire},\mathit{name}_1,\mathit{name}_2)} \mid \langle\!\langle \text{WLINES} \rangle\!\rangle_{(\mathit{wire},\mathit{name}_1,\mathit{name}_2)}$

- $$\begin{aligned} &\langle\!\langle \mathit{int}_1 \leftrightarrow \mathit{int}_2 : \text{🔔}\,\mathit{param}_1 \leftrightarrow \mathit{param}'_1, \ldots, \text{🔔}\,\mathit{param}_n \leftrightarrow \mathit{param}'_n \\ &\qquad\qquad \boxtimes\,\mathit{param}_{n+1} \leftrightarrow \mathit{param}'_{n+1}, \ldots, \boxtimes\,\mathit{param}_m \leftrightarrow \mathit{param}'_m \rangle\!\rangle_{(\mathit{wire},\mathit{name}_1,\mathit{name}_2)} = \\ &* \,[x_{\mathit{param}_1}, \ldots, x_{\mathit{param}_n}]\, \mathit{wire}_{\mathit{roleA}} \bullet \mathit{int}_1?\langle \mathit{id}_{\mathit{intra}}, \text{🔔}, x_{\mathit{param}_1}, \ldots, x_{\mathit{param}_n} \rangle. \\ &\qquad\qquad \mathit{name}_2 \bullet \mathit{int}_2!\langle \mathit{id}_{\mathit{intra}}, \text{🔔}, x_{\mathit{param}_1}, \ldots, x_{\mathit{param}_n} \rangle \\ &\mid * \,[x_{\mathit{param}'_{n+1}}, \ldots, x_{\mathit{param}'_m}]\, \mathit{wire}_{\mathit{roleB}} \bullet \mathit{int}_2?\langle \mathit{id}_{\mathit{intra}}, \boxtimes, x_{\mathit{param}'_{n+1}}, \ldots, x_{\mathit{param}'_m} \rangle. \\ &\qquad\qquad \mathit{name}_1 \bullet \mathit{int}_1!\langle \mathit{id}_{\mathit{intra}}, \boxtimes, x_{\mathit{param}'_{n+1}}, \ldots, x_{\mathit{param}'_m} \rangle \end{aligned}$$

5.2 Implementation of business roles

We introduce now the static aspects of the implementation, that is the implementation of the `ORCHESTRATION` part of a business role. Here we follow a bottom-up approach, i.e. before presenting the implementation of transitions, we introduce the implementation of variables, interaction parameters, expressions, assignments, send events and triggers.

Variables and interaction parameters. We implement every local variable (e.g., with name *lvar*) in the orchestration of a SRML business role as a pair of COWS *standard* variables, one representing the value of *lvar* before a transition, the other representing the value after the transition.

$$\langle\!\langle \text{LVARS}\,,\ \text{LVARS} \rangle\!\rangle = \langle\!\langle \text{LVARS} \rangle\!\rangle \mid \langle\!\langle \text{LVARS} \rangle\!\rangle$$

$$\langle\!\langle lvar : type \rangle\!\rangle = Var_{lvar} \mid Var_{lvar'}$$

Of course, at the end of a transition, we put $lvar = lvar'$.

Standard variables are services providing 'read' and 'write' operations. When the service variable is initialized (i.e. the first time the 'write' operation is used), an instance is created that is able to provide the value currently stored. When this value must be updated, the current instance is terminated and a new instance is created which stores the new value. Here is the specification:

$$\begin{aligned} Var_{lvar} \triangleq\, & [x_v, x_a]\, lvar \bullet o_{write}?\langle x_v, x_a\rangle. \\ & \quad [\hat{m}]\,(\hat{m}!\langle x_v, x_a\rangle \mid *\,[x, y]\,\hat{m}?\langle x, y\rangle. \\ & \qquad (y!\langle\rangle \mid [k]\,(*\,[y']\,lvar \bullet o_{read}?\langle y'\rangle.\{\!|y'!\langle x\rangle|\!\} \\ & \qquad\qquad \mid [x', y']\,lvar \bullet o_{write}?\langle x', y'\rangle\,.\,(\mathbf{kill}(k) \mid \{\!|\hat{m}!\langle x', y'\rangle|\!\}\,)\,)\,)\,) \end{aligned}$$

where the public partner name *lvar* has the same name of the variable. Service Var_{lvar} provides two operations: o_{read}, for getting the current value; o_{write}, for replacing the current value with a new one. To access the service, a user must invoke these operations by providing a communication endpoint for the reply and, in case of o_{write}, the value to be stored. The o_{write} operation can be invoked along the public partner *lvar*, which corresponds, the first time, to initialization of the variable. Every variable uses the delimited endpoint $\hat{m}$ in which to store the current value of the variable. This last feature is exploited to implement further o_{write} operations in terms of forced termination and re-instantiation. Delimitation $[k]$ is used to confine the effect of the kill activity to the current instance, while protection $\{\!|\_|\!\}$ avoids forcing termination of pending replies and of the invocation that will trigger the new instance.

Similarly, we implement every 🔔-parameter and ✉-parameter of every asynchronous interaction declared in a SRML business role, as the process $Var_{int.param}$ representing the variable with the name obtained by the concatenation of the interaction name (e.g., *int*) and the parameter name (e.g., *param*), as specified below.

- $\langle\!\langle \text{INTS}\ \ \text{INTS} \rangle\!\rangle = \langle\!\langle \text{INTS} \rangle\!\rangle \mid \langle\!\langle \text{INTS} \rangle\!\rangle$
- $\langle\!\langle \texttt{rcv}\ int$ 🔔 $\text{PARAMS} \rangle\!\rangle = \langle\!\langle \text{PARAMS} \rangle\!\rangle_{int}$
- $\langle\!\langle \textbf{snd}\ int$ 🔔 $\text{PARAMS} \rangle\!\rangle = \langle\!\langle \text{PARAMS} \rangle\!\rangle_{int}$
- $\langle\!\langle \texttt{r\&s}\ int$ 🔔 PARAMS_1 ✉ $\text{PARAMS}_2 \rangle\!\rangle = \langle\!\langle \text{PARAMS}_1\,,\ \text{PARAMS}_2 \rangle\!\rangle_{int} \mid State_{int}$
- $\langle\!\langle \texttt{s\&r}\ int$ 🔔 PARAMS_1 ✉ $\text{PARAMS}_2 \rangle\!\rangle = \langle\!\langle \text{PARAMS}_1\,,\ \text{PARAMS}_2 \rangle\!\rangle_{int} \mid State_{int}$

- $\langle\!\langle \texttt{ask}\ \mathit{int}(\text{TYPES}) : \mathit{type} \rangle\!\rangle \ = \ \mathbf{0}$
- $\langle\!\langle \texttt{rpl}\ \mathit{int}(\text{TYPES}) : \mathit{type} \rangle\!\rangle \ = \ \mathbf{0}$
- $\langle\!\langle \texttt{tll}\ \mathit{int}(\text{TYPES}) \rangle\!\rangle \ = \ \mathbf{0}$
- $\langle\!\langle \texttt{prf}\ \mathit{int}(\text{TYPES}) \rangle\!\rangle \ = \ \mathbf{0}$
- $\langle\!\langle \text{PARAMS}\,,\ \text{PARAMS} \rangle\!\rangle_{int} = \langle\!\langle \text{PARAMS} \rangle\!\rangle_{int} \mid \langle\!\langle \text{PARAMS} \rangle\!\rangle_{int}$
- $\langle\!\langle \mathit{param} : \mathit{type} \rangle\!\rangle_{int} = \mathit{Var}_{int.param}$

Each asynchronous conversational interaction *int* (i.e., having type either **`r&s`** or **`s&r`**) is also equipped with a state variable State_{int}, that is an auxiliary service, accessed through the endpoints $s\_int \bullet set$ and $s\_int \bullet get$, recording the state of the interaction. In particular, the admitted values are as follows: $\perp$ means that the interaction is not started, 🔔 means that the initial request has been performed, ✓ means that it has been committed, ✗ means that it has been cancelled, and ✞ means that it has been compensated. Formally, the term is as follows:

$$
\begin{aligned}
&\mathit{State}_{int} \triangleq s\_int \bullet get!\langle \perp \rangle \\
&\qquad\quad \mid s\_int \bullet set?\langle \text{🔔} \rangle.\,(s\_int \bullet get!\langle \text{🔔} \rangle \\
&\qquad\qquad \mid s\_int \bullet set?\langle \text{✉} \rangle.\,(s\_int \bullet get!\langle \text{✉} \rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid s\_int \bullet set?\langle \text{✗} \rangle.\ s\_int \bullet get!\langle \text{✗} \rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ + s\_int \bullet set?\langle \checkmark \rangle.\,(s\_int \bullet get!\langle \checkmark \rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid s\_int \bullet set?\langle \text{✞} \rangle.\ s\_int \bullet get!\langle \text{✞} \rangle)\,)\,)
\end{aligned}
$$

After a reply event ✉ the state can be set either to ✗ or ✓, and moreover the state ca be set to ✞ only after a commit. Notably, operations *get* and *set* are restricted to the scope of the considered components (see implementation of components in the previous section), thus to interact with the variable does not require to use any identifier.

Expressions. Here, we will use $\mathtt{x}$ to range over local variables (e.g. *lvar*, *lvar′*, ...) and interaction parameters (e.g. *int.param*), and $a, a_1, \ldots$ to indicate names of **`ask`** interactions. The implementation of an expression e is parameterized by two parameters: endpoint $\hat{m}$ returns the result of evaluating e, endpoint $\hat{n}$ is used whenever assigning the resulting value to a service variable.

To start with we show the implementation of an expression of the form $a(e_1, \ldots, e_k)$, i.e. of an **`ask`** interaction.

$$
\begin{aligned}
&\langle\!\langle a(e_1, \ldots, e_k) \rangle\!\rangle_{\hat{m},\hat{n}} \ = \\
&\quad [\hat{r}_1, \ldots, \hat{r}_n, \hat{r}]\,(\,\langle\!\langle e_1 \rangle\!\rangle_{\hat{r}_1,\hat{n}} \mid \ldots \mid \langle\!\langle e_k \rangle\!\rangle_{\hat{r}_k,\hat{n}} \\
&\qquad\qquad\qquad\quad \mid\ [x_1, \ldots, x_k]\,\hat{r}_1?\langle x_1, \hat{n} \rangle. \cdots .\hat{r}_k?\langle x_k, \hat{n} \rangle. \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\,a \bullet a!\langle x_1, \ldots, x_k, \hat{r} \rangle \mid [y]\,\hat{r}?\langle y \rangle.\hat{m}!\langle y, \hat{n} \rangle\,)\,)
\end{aligned}
$$

Now, the implementation of an expression e, that is not of the form $a(\bar{e})$, where $\mathtt{x}_1, \ldots, \mathtt{x}_n$ and $a_1(\bar{e}_1), \ldots, a_k(\bar{e}_k)$ are all the occurrences of ‘local variables’/‘interaction parameters' and **`ask`** interactions which do not occur as arguments of other **`ask`** interactions, is

$$\begin{aligned}\langle\!\langle e\rangle\!\rangle_{\hat{m},\hat{n}} = [\hat{r}_1,\ldots,\hat{r}_n,\hat{m}_1,\ldots,\hat{m}_k]\\ (\,\mathtt{x}_1\bullet o_{read}!\langle\hat{r}_1\rangle \mid \cdots \mid \mathtt{x}_n\bullet o_{read}!\langle\hat{r}_n\rangle \mid \langle\!\langle a_1(\bar{e}_1)\rangle\!\rangle_{\hat{m}_1,\hat{n}} \mid \ldots \mid \langle\!\langle a_k(\bar{e}_k)\rangle\!\rangle_{\hat{m}_k,\hat{n}}\\ \mid [x_1,\ldots,x_n,y_1,\ldots,y_k]\\ \hat{r}_1?\langle x_1\rangle.\cdots.\hat{r}_n?\langle x_n\rangle.\hat{m}_1?\langle y_1,\hat{n}\rangle.\cdots.\hat{m}_k?\langle y_k,\hat{n}\rangle.\\ \hat{m}!\langle e\cdot\{\mathtt{x}_i\mapsto x_i\}_{i\in\{1,..,n\}}\cdot\{a_j(\bar{e}_j)\mapsto y_j\}_{j\in\{1,..,k\}},\hat{n}\rangle\,)\end{aligned}$$

where $\{\mathtt{x}_i \mapsto x_i\}$ denotes substitution of $\mathtt{x}_i$ with x_i, and likewise $\{a_j(\bar{e}_j) \mapsto y_j\}$ denotes substitution of interactions $a_j(\bar{e}_j)$ with y_j. Of course, we are assuming that $\hat{m}$, $\hat{n}$, $\hat{r}_i$, $\hat{m}_j$, x_i and y_j are fresh.

Assignments. By exploiting the implementation of expressions, the implementation of an assignment is

$$\langle\!\langle \mathtt{x} = e\rangle\!\rangle_{\hat{n}} = \langle\!\langle e\rangle\!\rangle_{\mathtt{x}\bullet o_{write},\hat{n}}$$

where $\mathtt{x}$ is a variable *lvar*, a variable *lvar′* or a parameter *int.param*. The implementation is parameterized by an endpoint $\hat{n}$ that permits to receive an acknowledgment when the value resulting from evaluation of the expression on the right is assigned to the service variable on the left.

Thus, the implementation of initial assignments is

$$\langle\!\langle \mathrm{IASGS}_1 \ \wedge\ \mathrm{IASGS}_2\rangle\!\rangle_{\hat{n}} = [\hat{m}]\,(\,\langle\!\langle \mathrm{IASGS}_1\rangle\!\rangle_{\hat{m}} \mid \langle\!\langle \mathrm{IASGS}_2\rangle\!\rangle_{\hat{m}} \mid \hat{m}?\langle\rangle.\hat{m}?\langle\rangle.\hat{n}!\langle\rangle\,)$$

Instead, guarded assignments are implemented as the following COWS terms:

$$\langle\!\langle \mathrm{GASGS}_1 \wedge \mathrm{GASGS}_2\rangle\!\rangle_{\hat{n}} = [\hat{m}]\,(\,\langle\!\langle \mathrm{GASGS}_1\rangle\!\rangle_{\hat{m}} \mid \langle\!\langle \mathrm{GASGS}_2\rangle\!\rangle_{\hat{m}} \mid \hat{m}?\langle\rangle.\hat{m}?\langle\rangle.\hat{n}!\langle\rangle\,)$$

$$\langle\!\langle c \ \supset\ \mathrm{ASG}\rangle\!\rangle_{\hat{n}} = [\hat{m}]\,(\langle\!\langle c\rangle\!\rangle_{\hat{m},\hat{n}} \mid \hat{m}?\langle\mathsf{true},\hat{n}\rangle.\langle\!\langle \mathrm{ASG}\rangle\!\rangle_{\hat{n}} \ +\ \hat{m}?\langle\mathsf{false},\hat{n}\rangle.\hat{n}!\langle\rangle)$$

Guarded sends. In the previous section, we have seen that the implementation function for transitions is parameterized by an environment environment $\mathcal{E}$ storing information about types and parameters of interactions. More specifically, $\mathcal{E}$ is a couple of functions $\langle\mathcal{E}_t,\mathcal{E}_p\rangle$:

- $\mathcal{E}_t$ returns the type of an interaction: that is, given the interaction *int*, $\mathcal{E}_t(\mathit{int}) \in$ {**`rcv`**, **`snd`**, **`r&s`**, **`s&r`**, **`ask`**, **`rpl`**, **`tll`**, **`prf`**};
- $\mathcal{E}_p$, according to the type of an interaction, returns the list of parameters of the interaction. Thus, according to the type of a given the interaction *int*, we can use notations $\mathcal{E}_p($🔔$, \mathit{int})$, $\mathcal{E}_p($✉$, \mathit{int})$ and $\mathcal{E}_p(\mathit{int})$ to indicate the list of its parameters.

The environment will be used here as parameter of the implementation function for send events. In the sequel, we will sometimes omit the environment when it is not used. Given a list L, we will use notation $\sharp L$ to indicate the cardinality of L, and $L\downarrow_i$ (with $1 \leq i \leq \sharp L$) to denote the 'projection' of L, i.e. the ith element of L.

Here, without loss of generality, we assume that assignments within a **`sends`** block always precede sends events. This, together with the fact that a composition of guarded sends are COWS term executed sequentially, guarantees that the standard variables modelling interaction parameters are updated before the execution of the corresponding send events.

$$\begin{aligned}&\langle\!\langle \mathrm{GSENDS}_1 \wedge \mathrm{GSENDS}_2 \rangle\!\rangle^{\mathcal{E}}_{\hat{n}} = \\ &\qquad [\hat{m}_1, \hat{m}_2]\,(\, \langle\!\langle \mathrm{GSENDS}_1 \rangle\!\rangle^{\mathcal{E}}_{\hat{m}_1} \mid \langle\!\langle \mathrm{GSENDS}_2 \rangle\!\rangle^{\mathcal{E}}_{\hat{m}_2} \;\mid\; \hat{m}_1?\langle\rangle.\hat{m}_2?\langle\rangle.\hat{n}!\langle\rangle\,)\end{aligned}$$

$$\langle\!\langle c \;\supset\; int\ \mathrm{ET} \rangle\!\rangle^{\mathcal{E}}_{\hat{n}} = [\hat{m}]\,(\langle\!\langle c \rangle\!\rangle_{\hat{m},\hat{n}} \mid \hat{m}?\langle \mathsf{true}, \hat{n}\rangle.\langle\!\langle int\ \mathrm{ET} \rangle\!\rangle^{\mathcal{E}}_{\hat{n}} \;+\; \hat{m}?\langle \mathsf{false}, \hat{n}\rangle.\hat{n}!\langle\rangle)$$

The assignments are implemented as before, while the send events as follows:

- $$\begin{aligned}\langle\!\langle int\ \text{🔔} \rangle\!\rangle^{\mathcal{E}}_{\hat{n}} = [\hat{r}_1, \ldots, \hat{r}_n]\,(\, &\mathtt{x}_1 \bullet o_{read}!\langle \hat{r}_1 \rangle \mid \cdots \mid \mathtt{x}_n \bullet o_{read}!\langle \hat{r}_n \rangle \\ &\mid [x_1, \ldots, x_n]\, \hat{r}_1?\langle x_1 \rangle. \cdots .\hat{r}_n?\langle x_n \rangle. \\ &\qquad (\, int \bullet int!\langle id_{intra}, \text{🔔}, x_1, \ldots, x_n \rangle \mid \hat{n}!\langle\rangle\,)\,)\end{aligned}$$

  if $\mathcal{E}_p(\text{🔔}, int) = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle$ and $\mathcal{E}_t(int) = \mathbf{snd}$.

- $$\begin{aligned}\langle\!\langle int\ \text{🔔} \rangle\!\rangle^{\mathcal{E}}_{\hat{n}} = [\hat{r}_1, \ldots, \hat{r}_n]\,(\, &\mathtt{x}_1 \bullet o_{read}!\langle \hat{r}_1 \rangle \mid \cdots \mid \mathtt{x}_n \bullet o_{read}!\langle \hat{r}_n \rangle \\ &\mid [x_1, \ldots, x_n]\, \hat{r}_1?\langle x_1 \rangle. \cdots .\hat{r}_n?\langle x_n \rangle.\ s\_int \bullet get?\langle \bot \rangle. \\ &\qquad (\, s\_int \bullet set!\langle \text{🔔} \rangle \mid int \bullet int!\langle id_{intra}, \text{🔔}, x_1, \ldots, x_n \rangle \mid \hat{n}!\langle\rangle\,)\,)\end{aligned}$$

  if $\mathcal{E}_p(\text{🔔}, int) = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle$ and $\mathcal{E}_t(int) = \mathbf{s\&r}$.

- $$\begin{aligned}\langle\!\langle int\ \boxtimes \rangle\!\rangle^{\mathcal{E}}_{\hat{n}} = [\hat{r}_1, \ldots, \hat{r}_n]\,(\, &\mathtt{x}_1 \bullet o_{read}!\langle \hat{r}_1 \rangle \mid \cdots \mid \mathtt{x}_n \bullet o_{read}!\langle \hat{r}_n \rangle \\ &\mid [x_1, \ldots, x_n]\, \hat{r}_1?\langle x_1 \rangle. \cdots .\hat{r}_n?\langle x_n \rangle.\ s\_int \bullet get?\langle \text{🔔} \rangle. \\ &\qquad (\, s\_int \bullet set!\langle \boxtimes \rangle \mid int \bullet int!\langle id_{intra}, \boxtimes, x_1, \ldots, x_n \rangle \mid \hat{n}!\langle\rangle\,)\,)\end{aligned}$$

  if $\mathcal{E}_p(\boxtimes, int) = \langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle$.

- $\langle\!\langle int\ \checkmark \rangle\!\rangle^{\mathcal{E}}_{\hat{n}} \;=\; s\_int \bullet get?\langle \boxtimes \rangle.\,(\, s\_int \bullet set!\langle \checkmark \rangle \mid int \bullet int!\langle id_{intra}, \checkmark \rangle \mid \hat{n}!\langle\rangle\,)$

- $\langle\!\langle int\ \text{✘} \rangle\!\rangle^{\mathcal{E}}_{\hat{n}} \;=\; s\_int \bullet get?\langle \boxtimes \rangle.\,(\, s\_int \bullet set!\langle \text{✘} \rangle \mid int \bullet int!\langle id_{intra}, \text{✘} \rangle \mid \hat{n}!\langle\rangle\,)$

- $\langle\!\langle int\ \text{✞} \rangle\!\rangle^{\mathcal{E}}_{\hat{n}} \;=\; s\_int \bullet get?\langle \checkmark \rangle.\,(\, s\_int \bullet set!\langle \text{✞} \rangle \mid int \bullet int!\langle id_{intra}, \text{✞} \rangle \mid \hat{n}!\langle\rangle\,)$

- $\langle\!\langle int\ \Lsh\ e \rangle\!\rangle^{\mathcal{E}}_{\hat{n}} \;=\; [\hat{m}, \hat{r}]\,(\, \langle\!\langle e \rangle\!\rangle_{\hat{m},\hat{r}} \mid [x]\, \hat{m}?\langle x, \hat{r}\rangle.\,(\, x_{int.ret}!\langle id_{intra}, x \rangle \mid \hat{n}!\langle\rangle\,)\,)$

- $$\begin{aligned}\langle\!\langle int(e_1, \ldots\ e_k) \rangle\!\rangle^{\mathcal{E}}_{\hat{n}} \;=\; [\hat{m}_1, \ldots, \hat{m}_k, \hat{r}]\,(\, &\langle\!\langle e_1 \rangle\!\rangle_{\hat{m}_1,\hat{r}} \mid \ldots \mid \langle\!\langle e_k \rangle\!\rangle_{\hat{m}_k,\hat{r}} \\ &\mid [x_1, \ldots, x_k]\, \hat{m}_1?\langle x_1, \hat{r}\rangle.\ \ldots \hat{m}_k?\langle x_k, \hat{r}\rangle. \\ &\qquad [\widehat{ret}]\,(\, int \bullet int!\langle id_{intra}, x_1, \ldots, x_k, \widehat{ret} \rangle \mid \widehat{ret}?\langle\rangle.\,\hat{n}!\langle\rangle\,)\,)\end{aligned}$$

Basically, a send event of the form *int* ET is implemented as a term that firstly retrieves the values of the associated parameters, if the interaction has someone of them, then properly sets the state variable of the involved interaction and performs an invoke activity along the endpoint $int \bullet int$ by sending the intra-module identifier, the event type and the parameter values. In the last two cases the interaction must be synchronous. In particular, $int\ \Lsh\ e$ is performed by the continuation of an **rpl** interaction to reply to an **ask** interaction. Thus, its implementation evaluates the expression *e* and then returns its value along a private channel previously received (see implementation of **`ask`** interaction) and stored in $x_{int.ret}$ (see implementation of triggers). Instead, $int(e_1, \ldots e_k)$ is the send event of a **`tll`** interaction. Therefore, a (fresh) endpoint $\widehat{ret}$ is sent and an acknowledgement is waited for (like in the implementation of **`ask`** interaction).

Triggers. In order to guarantee progress properties, when a transition with a trigger is selected by the scheduler, it has a given interval of time to synchronize with the other party and proceed. After the time is elapsed, the turn of execution passes to another transition. This way, transitions will not get stuck forever waiting on a trigger event. Triggers can be naturally modelled in COWS by exploiting the addition of 'timed' activities. Timed activities have been introduced in [20], since it is not known to what extent timed computation can be reduced to untimed forms of computation [31]. Specifically, COWS is extended with a WS-BPEL-like *wait* activity of the form 🕒_e, that suspends the execution of the invoking service until the time interval whose duration is specified as an argument has elapsed and can be used as a guard for the choice operator.

Therefore, triggers are implemented as the following COWS terms:

- $$\begin{aligned}\langle\!\langle int\ \text{🔔}\rangle\!\rangle^{\mathcal{E}}_{(comp,\hat{n})} = [x_1, \ldots, x_k]\ & \\ (\,comp \bullet int?\langle id_{intra}, \text{🔔}, x_1, \ldots, x_k\rangle. & \\ [\hat{m}]\,(\,\langle\!\langle \mathcal{E}_p(\text{🔔}, int) \downarrow_1 = x_1 \rangle\!\rangle_{\hat{m}} & \\ \mid \hat{m}?\langle\rangle.(\cdots \mid \hat{m}?\langle\rangle.\langle\!\langle \mathcal{E}_p(\text{🔔}, int) \downarrow_k = x_k \rangle\!\rangle_{\hat{m}} & \\ \mid \hat{m}?\langle\rangle.(s\_int \bullet set!\langle \text{🔔}\rangle \mid \hat{n}!\langle \mathsf{true}\rangle) \ldots) & \\ +\ \text{🕒}_{\delta}.\,\hat{n}!\langle \mathsf{false}\rangle\,) &\end{aligned}$$

  where $k = \sharp\mathcal{E}_p(\text{🔔}, int)$; this case deals with $\mathcal{E}_t(int) =$ **`rcv`** and $\mathcal{E}_t(int) =$ **`r&s`**.

- $$\begin{aligned}\langle\!\langle int\ \boxtimes\rangle\!\rangle^{\mathcal{E}}_{(comp,\hat{n})} = [x_1, \ldots, x_k]\ & \\ (\,comp \bullet int?\langle id_{intra}, \boxtimes, x_1, \ldots, x_k\rangle. & \\ [\hat{m}]\,(\,\langle\!\langle \mathcal{E}_p(\boxtimes int) \downarrow_1 = x_1 \rangle\!\rangle_{\hat{m}} & \\ \mid \hat{m}?\langle\rangle.(\cdots \mid \hat{m}?\langle\rangle.\langle\!\langle \mathcal{E}_p(\boxtimes, int) \downarrow_k = x_k \rangle\!\rangle_{\hat{m}} & \\ \mid \hat{m}?\langle\rangle.(s\_int \bullet set!\langle \boxtimes\rangle \mid \hat{n}!\langle \mathsf{true}\rangle) \ldots) & \\ +\ \text{🕒}_{\delta}.\,\hat{n}!\langle \mathsf{false}\rangle\,) &\end{aligned}$$

  where $k = \sharp\mathcal{E}_p(\boxtimes, int)$; this case deals with $\mathcal{E}_t(int) =$ **`s&r`**.

- $$\begin{aligned}\langle\!\langle int\ \mathrm{ET}\rangle\!\rangle^{\mathcal{E}}_{(comp,\hat{n})} = comp \bullet int?\langle id_{intra}, \mathrm{ET}\rangle.(s\_int \bullet set!\langle \mathrm{ET}\rangle \mid \hat{n}!\langle \mathsf{true}\rangle) & \\ +\ \text{🕒}_{\delta}.\,\hat{n}!\langle \mathsf{false}\rangle &\end{aligned}$$

  where $\mathrm{ET} \in \{\checkmark, \text{✗}, \text{✞}\}$.

- $$\begin{aligned}&\langle\!\langle int(param_1, \ldots, param_k)\rangle\!\rangle^{\mathcal{E}}_{(comp,\hat{n})} = \\ &\quad [x_1, \ldots, x_k, x_{ret}] \\ &\quad (\,comp \bullet int?\langle id_{intra}, x_1, \ldots, x_k, x_{ret}\rangle. \\ &\qquad [\hat{m}]\,(\,\langle\!\langle param_1 = x_1 \rangle\!\rangle_{\hat{m}} \mid \hat{m}?\langle\rangle. \\ &\qquad\qquad (\cdots \mid \hat{m}?\langle\rangle.\langle\!\langle param_k = x_k \rangle\!\rangle_{\hat{m}} \mid \hat{m}?\langle\rangle.\,\hat{n}!\langle \mathsf{true}\rangle\,) \ldots) \\ &\quad +\ \text{🕒}_{\delta}.\,\hat{n}!\langle \mathsf{false}\rangle\,)\end{aligned}$$

this case deals with $\mathcal{E}_t(int) = \mathtt{\textbf{rpl}}$ and $\mathcal{E}_t(int) = \mathtt{\textbf{prf}}$.

Basically, triggers are implemented in terms performing a receive activity along the endpoint $comp \bullet int$ and, in case the involved interaction has some parameters, then the received values are assigned to them. In case of asynchronous conversational interactions, the state variable is also properly updated. Moreover, if the trigger event associated to the transition does not occur within δ time units, then a "skip-event" $\hat{n}!\langle \mathsf{false} \rangle$ is performed, and the turn of execution passes to another transition. Notably, the implementation function of triggers has as subscript a pair $(comp, \hat{n})$; this permits distinguishing the implementation of triggers from that of send events. Notice also that here along endpoint $\hat{n}$ are transmitted boolean values indicating the success/unsuccess of the turn of execution.

We could implement SRML trigger without exploiting timed constructs, but relying on the prioritized semantics of COWS and an additional fairness assumption for the semantics of the parallel composition. Anyway, the understanding of the resulting implementation results to be more tricky.

Transitions. The implementation of transitions is given by the following COWS terms:

$$\langle\!\langle \mathrm{TRANS}_1\ \mathrm{TRANS}_2 \rangle\!\rangle^{\mathcal{E}}_{comp} \;=\; \langle\!\langle \mathrm{TRANS}_1 \rangle\!\rangle^{\mathcal{E}}_{comp} \mid \langle\!\langle \mathrm{TRANS}_2 \rangle\!\rangle_{comp}\mathcal{E}$$

where if TRANS is

`transition` *tr*
 `triggeredBy` TRIGS
 `guardedBy` c'
 `effects` GASGS
 `sends` GSENDS

then, if TRIGS $= c$, its implementation $\langle\!\langle \mathrm{TRANS} \rangle\!\rangle^{\mathcal{E}}_{comp}$ is given by

$$\begin{array}{l}
*\,[\hat{m}, \hat{n}]\,(\, st\hat{art}?\langle\rangle.\langle\!\langle c \rangle\!\rangle_{\hat{n},\hat{m}} \\
\qquad\quad \mid \hat{m}?\langle \mathsf{true}, \hat{n} \rangle.(\, \langle\!\langle c' \rangle\!\rangle_{\hat{m},\hat{n}} \mid \hat{m}?\langle \mathsf{true}, \hat{n} \rangle.(\, \langle\!\langle \mathrm{GASGS} \rangle\!\rangle_{\hat{n}} \mid \hat{n}?\langle\rangle. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\, \langle\!\langle \mathrm{GSENDS}' \rangle\!\rangle^{\mathcal{E}}_{\hat{n}} \mid \hat{n}?\langle\rangle.e\hat{nd}!\langle\rangle\,)\,) \\
\qquad\qquad\qquad\qquad\qquad\quad +\; \hat{m}?\langle \mathsf{false}, \hat{n} \rangle.e\hat{nd}!\langle\rangle\,) \\
\qquad\quad +\; \hat{m}?\langle \mathsf{false}, \hat{n} \rangle.e\hat{nd}!\langle\rangle\,)
\end{array}$$

Here, GSENDS$'$ denotes GSENDS enriched with those assignments of form $\mathtt{x} = \mathtt{x}'$ that are needed to align the values of the pair of standard variables corresponding to a given local variable or parameter after a transition. Basically, after the transition has acquired the lock, it evaluates the triggering condition c and the guard c'. If both of them hold true, it performs firstly the assignments within the **`effects`** block, then the activities within the **`sends`** block, and finally releases the lock. If one of c and c' does not hold, it immediately releases the lock.

More generally, however, it is the case when the trigger is not a simple condition (i.e. TRIGS $\neq c$). In this case, the transition cannot take place more than once. Thus, the implementation is

$$\begin{array}{l}
[k_{all}] * [\hat{n}] \, (\, \hat{start}?\langle\rangle.\langle\!\langle \mathrm{TRIGS} \rangle\!\rangle^{\mathcal{E}}_{(comp,\hat{n})} \\
\qquad\qquad | \; \hat{n}?\langle \mathsf{true} \rangle.[\hat{m}] \, (\, \langle\!\langle c' \rangle\!\rangle_{\hat{m},\hat{n}} \\
\qquad\qquad\qquad\qquad\qquad | \; \hat{m}?\langle \mathsf{true}, \hat{n} \rangle.(\, \langle\!\langle \mathrm{GASGS} \rangle\!\rangle_{\hat{n}} \mid \hat{n}?\langle\rangle.(\, \langle\!\langle \mathrm{GSENDS}' \rangle\!\rangle^{\mathcal{E}}_{\hat{n}} \mid \hat{n}?\langle\rangle.K_{tr} \,) \,) \\
\qquad\qquad\qquad\qquad\qquad\; + \; \hat{m}?\langle \mathsf{false}, \hat{n} \rangle.K_{tr} \,) \\
\qquad\qquad\; + \; \hat{n}?\langle \mathsf{false} \rangle.\hat{end}!\langle\rangle \,)
\end{array}$$

where K_{tr} stands for the COWS term $\mathbf{kill}(k_{all}) \mid \{\!|\hat{end}!\langle\rangle|\!\}$. In fact, since any interaction event may occur at most once, we associate a killer label k_{all} to each transition in order to disable the transition from the scheduling after its first execution.

6 Conclusion

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 [16], 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 [19, 18, 12, 10, 11, 32]), 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 [25], UML4SOA, an UML-based domain-specific language, is used for modelling SOA artefacts, while WS-BPEL, Java and Jolie[4] 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 [30], 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 [13], 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 [28]). Many of these efforts aim at formalizing its semantics using Petri nets [28, 24], 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 [23] 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 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.

---

[4] `http://www.jolie-lang.org/`

6. M. Beisiegel et al. 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`.
7. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.
8. L. Bocchi, J.L. Fiadeiro, A. Lapadula, R. Pugliese, and F. Tiezzi. From Architectural to Behavioural Specification of Services. In *FESCA*, ENTCS. Elsevier, 2009. To appear.
9. L. Bocchi, Y Hong, A Lopes, and J.L. Fiadeiro. From BPEL to SRML: A Formal Transformational Approach. In *WSFM*, volume 4937 of *LNCS*, pages 92–107. Springer, 2007.
10. M. Boreale, Roberto Bruni, Rocco De Nicola, and Michele Loreti. Sessions and pipelines for structured service programming. In *FMOODS*, volume 5051 of *LNCS*, pages 19–38. Springer, 2008.
11. 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.
12. 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.
13. 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*, 2009. To appear.
14. 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.
15. J.L. Fiadeiro, A. Lopes, and L. Bocchi. A formal approach to service component architecture. In *WSFM*, volume 4184 of *LNCS*, pages 193–213. Springer, 2006.
16. 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/`.
17. I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., 2003.
18. 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.
19. C. Laneve and G. Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.
20. A. Lapadula, R. Pugliese, and F. Tiezzi. C◷WS: A timed service-oriented calculus. In *ICTAC*, volume 4711 of *LNCS*, pages 275–290. Springer, 2007.
21. 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.
22. 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.
23. A. Lapadula, R. Pugliese, and F. Tiezzi. Service discovery and negotiation with COWS. In *WWV*, volume 200 of *ENTCS*, pages 133–154. Elsevier, 2008.
24. N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. In *WSFM*, volume 4937 of *LNCS*, pages 77–91. Springer, 2008.
25. P. Mayer, A. Schroeder, and N. Koch. MDD4SOA: Model-Driven Service Orchestration. In *EDOC*, pages 203–212. IEEE Computer Society Press, 2008.
26. R. Milner. *Communicating and Mobile Systems: The piCalculus*. Cambridge University Press, 1999.
27. 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`.

28. 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`.
29. D. Prandi and P. Quaglia. Stochastic COWS. In *ICSOC*, volume 4749 of *LNCS*, pages 245–256. Springer, 2007.
30. 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.
31. R.J. van Glabbeek. On specifying timeouts. *ENTCS*, 162:173–175, 2006.
32. H.T. Vieira, L. Caires, and J. Costa Seco. The conversation calculus: A model of service-oriented computation. In *ESOP*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.