Open access • Book Chapter • DOI:10.1007/11841197_13

A Formal Approach to Service Component Architecture — Source link

José Luiz Fiadeiro, Antónia Lopes, Laura Bocchi

Institutions: University of Leicester, University of Lisbon

Published on: 08 Sep 2006 - Web Services and Formal Methods

Topics: Orchestration (computing), Formal methods, Middleware (distributed applications), Service Component Architecture and Software architecture

A Formal Approach to Service Component Architecture[†]

José Luiz Fiadeiro[1], Antónia Lopes[2] and Laura Bocchi[1]

[1] Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
{bocchi,jose}@mcs.le.ac.uk

[2] Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa, PORTUGAL
mal@di.fc.ul.pt

Abstract. We report on a formal framework being developed within the SENSORIA project for supporting service-oriented modelling at high levels of abstraction, i.e. independently of the hosting middleware and hardware platforms, and the languages in which services are programmed. More specifically, we give an account of the concepts and techniques that support the composition model of SENSORIA, i.e. the mechanisms through which complex applications can be put together from simpler components, including modelling primitives for the orchestration of components and the definition of external interfaces.

1 Introduction

One of the goals of SENSORIA – an IST-FET Integrated Project on *Software Engineering for Service-Oriented Overlay Computers* – is to define a formal framework that can support a Reference Modelling Language (SRML) that operates at the higher levels of abstraction of "business" or "domain" architectures. The term "service-oriented" is taken within SENSORIA in a broad sense that encompasses the general principles and techniques either available or envisioned for Web Services [1], as well as other manifestations such as Grid Computing [11]. The aim is to develop concepts and techniques that are independent of what are sometimes called "global computers", i.e. the technologies that provide the middleware infrastructure over which services can be deployed, published and discovered. In this sense, our aims are in tune with the goal of the industrial consortium that is developing the Service Component Architecture (SCA) [14]. Like in SCA, we are aiming to support ways through which

> *[…] relatively coarse-grained business components can be exposed as services, with well-defined interfaces and contracts, removing or abstracting middleware programming model dependencies from business logic.*

[†] This work was partially supported through the IST-2005-16004 Integrated Project *SENSORIA: Software Engineering for Service-Oriented Overlay Computers*, and the Marie-Curie TOK-IAP MTK1-CT-2004-003169 *Leg2Net: From Legacy Systems to Services in the Net.*

The main concern of SCA in developing this middleware-independent layer is to provide an open specification "allowing multiple vendors to implement support for SCA in their development tools and runtimes". This is why SCA offers specific support for a variety of component implementation and interface types such as BPEL processes with WSDL interfaces, and Java classes with corresponding interfaces. Our work explores a complementary direction: our research aims for a mathematical semantics of a Service Component Architecture that can provide a uniform model of service behaviour in a way that is independent of the languages and technologies used for programming and deploying services. Besides SCA, we also take into account recent advances on Web Services such as [1,6], and stay as close as possible to the terminology that is being adopted in the area.

More specifically, we develop a minimalist formal framework based on a core set of primitives and a language that is "small" enough to be formalised relatively easily and yet "powerful" enough to capture the essence of a new modelling paradigm centred on services. In this paper, we report on some of the efforts made so far in the development of this language by presenting fragments of its composition model, what we call SRML-P: the techniques through which one can model individual business components and interconnect them to build complex applications in a service-oriented way. A more detailed account of our approach is available in [9]. Issues related with dynamic configuration, such as service discovery and binding, are also being addressed over the model that we outline here.

In Section 2, we provide an overview of the composition model that we support in SRML-P. In Section 3, we present the primitives that we use for describing interactions. In Section 4, we discuss the modelling of components as orchestrations of interactions maintained with other parties. In Section 5, we show how external interfaces can be described in terms of sentences of a formal logic that model conversations. In Section 6, we discuss the way components can be wired to each other and to external interfaces in order to produce modules. In the concluding remarks, we point to other aspects that are being investigated and discuss the way we are taking this programme forwards. For illustration, we use a typical procurement business process involving a supplier, a warehouse and a local stock.

2 The Composition Model

SRML-P provides a language for modelling composite services, understood as services whose business logic involves a number of interactions among more elementary service components as well the invocation of services provided by other parties. As in SCA, interactions are supported on the basis of service interfaces defined in a way that is "independent of the hardware platform, the operating system, hosting middleware and the programming language used to implement the service" [14].

Central to the composition model is the notion of *service component*, or component for short. In SRML-P, a component is a computational unit that is modelled by means of an execution pattern involving a number of interactions that it can maintain with other parties. We refer to the execution pattern of a component as an *orchestration element*, or *orchestration* for short. The W3C Web Services Glossary[1] defines orchestration as

[…] the sequence and conditions in which one Web service invokes other Web services in order to realize some useful function.

In our context, the orchestration of the service provided by a module is the composition of the orchestrations defined within the components and the way they are wired together.

Each orchestration element is defined independently of the language in which the component is programmed and the platform in which it is deployed; it may be a BPEL process, a Java program, a wrapped-up legacy system, inter alia. In addition, the orchestration is independent of the specific parties that are actually interconnected with the component in any given run-time configuration; a component is totally independent in the sense that it does not invoke services of any specific co-party (i.e. an external service or another component) – it just offers an interface of two-way interactions in which it can participate.

As such, service components do not provide any business logic: the units of business logic are *modules* that use such components to provide services when they are interconnected with a number of other parties offering a number of required services. In a SRML-P module, both the provided services and those required from other parties are modelled as *external interfaces*, or interfaces for short. Each such interface specifies a stateful interaction between a service component and the corresponding party, i.e. SRML-P supports both "syntactic" and "behavioural" interfaces.

The external interface offered by a module to be used by clients, what in SCA corresponds to an "entry point", specifies constraints on the interactions that the module supports as a service provider such as the order in which it expects invocations or deadlines for the user to commit; it is the responsibility of the clients to adhere to these protocols, meaning that the provider may not be ready to engage in interactions that are not according to the specified constraints. Other properties are specified that any client may expect such as pledges on given parameters of the delivered services. The external interfaces to services required from other parties, what in SCA corresponds to "external services", specify the conversations that the module expects relative to each party.

Service components and external interfaces are connected to each other within modules through *internal wires* that bind the interactions that both parties declare to support. In SRML-P, all names are local, which implies that any interconnection needs to be made explicit through a wire that binds the names used locally in each party. The idea is to support reuse of both service components and external interfaces, thus facilitating the process of designing business applications. The coupling of service components within modules can be seen to be tight and performed at design time, reflecting the fact that they offer an (atomic) unit of business logic.

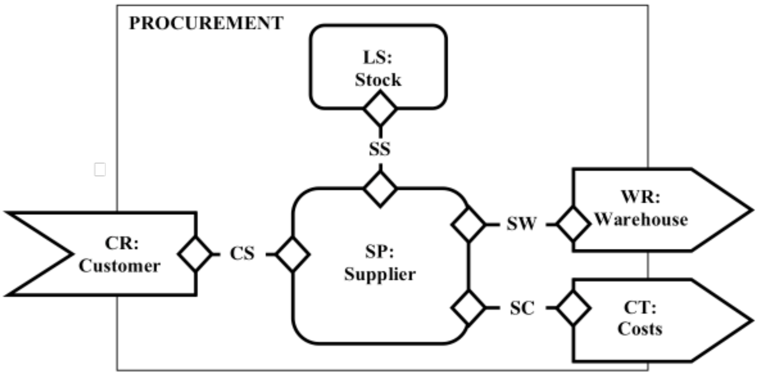
The table below establishes a relationship between the terminology that we use in SRML-P and the W3C Web Services Glossary. However, as already mentioned, in

[1] http://www.w3.org/TR/ws-gloss/

SRML-P we are aiming for higher-levels of abstraction in service-oriented modelling, which explains why this relationship is not a one-to-one mapping.

W3C	SRML-P	Relationship
Service	Module	A **module** defines how a certain **service** is provided through the coordination of a set of internal components and external services.
Service Description	External Interface (Provides/Requires)	**External Interfaces** correspond to **service descriptions** that include the interface and the interactive behaviour of the services provided/required by a module.
Orchestration	Orchestration	In SRML-P, **orchestration** is spread among all the components within a module.

In order to illustrate how applications are modelled in SRML-P, we use a typical procurement business process involving a supplier, a warehouse, a local stock, and a price look-up facility. The decision to make the local stock a component of the module reflects the tight coupling that exists with the supplier in business terms. The choice of warehouse should probably be made at run-time, for instance taking into account properties of the customer like its location, which justifies that it is represented in the module as an external interface. The price look-up facility is also a good example of an external service that may be shared among several suppliers.



This module declares two components: *SP* and *LS*. Components are typed by what we call *business roles*, which are discussed in Section 4; in this case, *SP* plays the business role of *Supplier* and *LS* of *Stock*. Three external interfaces are declared: one provides-interface – *CR* – and two requires-interfaces – *WR* and *CT*. Each such interface is typed by what we call a *business protocol* as discussed in Section 5; in the example, the business protocols are *Customer, Warehouse* and *Costs*, respectively. Finally, four wires connect components and interfaces: *CS*, *SS, SW* and *SC*. Each wire is labelled by an *interaction protocol* as discussed in Section 6; the labelling of wires is not easily depicted in figures such as above and is normally given in the textual definition of the module only. More details on the notion of module, including an algebraic semantics, can be found in [9].

3 The Language of Interactions

In this section, we provide a short account of the primitives that are being defined for describing interactions, taking into account proposals that have been made for Web-Services [4], in orchestration languages such as ORC [13], and in calculi such as Sagas [5]. However, because our aim is to support an abstract and declarative style of specification, our language will use some of these concepts (e.g. compensations, pledges, locking-properties, deadlines and timeouts) in a somewhat different way.

In SRML, we distinguish several types of interactions as shown in the table below. Interactions involve two parties and can be in both directions, i.e. they can be conversational. Interactions are described from the point of view of the party in which they are declared, i.e. "receive" means invocations received by the party and sent by the co-party, and "send" means invocations made by the party. Interactions can be synchronous, implying that the party waits for the co-party to reply or complete, or asynchronous, in which case the party does not block. The reason for choosing to have non-blocking asynchronous interactions is that we can leave it to the orchestration of the components to engage or not in other interactions while waiting for a reply.

r&s	The interaction is initiated by the co-party, which expects a reply. The co-party does not block while waiting for the reply.
s&r	The interaction is initiated by the party and expects a reply from its co-party. While waiting for the reply, the party does not block.
rcv	The co-party initiates the interaction and does not expect a reply.
snd	The party initiates the interaction and does not expect a reply.
ask	The party synchronises with the co-party to obtain data.
rpl	The party synchronises with the co-party to transmit data.
tll	The party requests the co-party to perform an operation and blocks.
prf	The party performs an operation and frees the co-party that requested it.

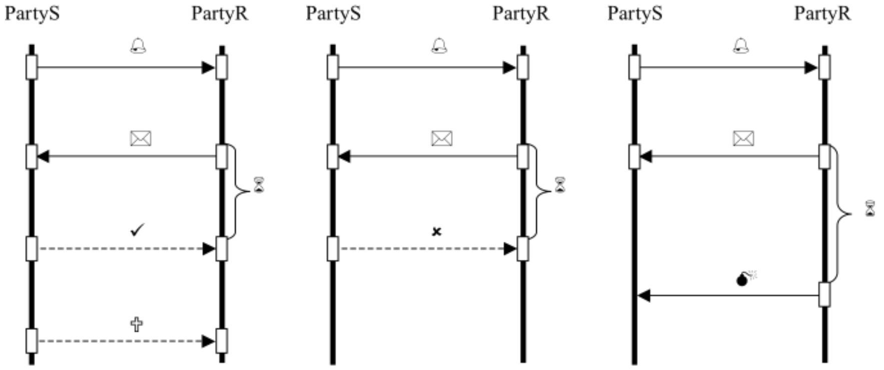
Notice that r&s and s&r interactions are durative/conversational. We distinguish several events that can occur during such interactions:

`interaction🔔`	The event of initiating *interaction.*
`interaction✉`	The reply-event of *interaction.*
`interaction✓`	The commit-event of *interaction.*
`interaction✗`	The cancel-event of *interaction.*
`interaction💣`	The deadline-event of *interaction.*
`interaction✞`	The revoke-event of *interaction.*

Further to these events, each such interaction may have an associated *pledge* – a condition that is guaranteed to hold from the moment a positive reply-event occurs until either the commit, the cancel or the deadline-event happens, whichever comes first. We denote this condition by *interaction*⌛. A reply-event *interaction*✉ is positive iff the distinguished Boolean parameter *Reply* is true.

The sequence diagrams below illustrate the intuitive semantics of these primitives when a pledge is offered. In the case on the left, the initiator commits to the pledge; a revoke may occur later on compensating the effects of *interaction*. In the case in the middle, there is a cancellation; in this situation, a revoke is not available. In the case on the right, the deadline-event occurs without a commit or cancel having occurred; this implies that no further events for that interaction will occur. In Section 4, we give examples of the usage of these primitives.

Events can be referred to from the point of view of the party that initiate them, in which case we use the notation *event!*, or the party that receives them, in which case we use *event?*. Events occur during state transitions in both parties involved in the interaction and require that the parties are available to perform the event; in other words, events are blocking in the sense that a party wishing to issue *event!* needs to wait for its co-party to be able to perform *event?*.



Interactions can have parameters for transmitting data when they are initiated, declared as 🔔, and for carrying a reply, declared as ✉. Notice that the boolean ✉-parameter *Reply* is always available, indicating if the reply is positive Only the additional parameters required for carrying data associated with the reply need to be declared. Key parameters, marked as 🔑, can also be declared which are used for generating different instances of a given class of events.

We assume that there are a number of "global" interactions provided by "the environment" such as time-related activities. This is necessary for parties to have some common understanding of issues like deadlines. In this paper, we will make use of the interaction *alertDate*, which is initiated by a party with a 🔑-parameter – *Ref* of type *string*, and a 🔔-parameter – *Interval* of type *date*. The agreed meaning is that the environment publishes *alertDate*✉ when *Interval* units of time have elapsed. Any party can subscribe to that event.

We make use of a number of connectives to formulate behavioural properties, examples of which are given throughout the paper. The following table summarises the intuitive meaning and the way some of them can be formulated in a branching time logic with linear past (see [12]).

a **before** b	If *b* holds then *a* must have been true.	$\mathbf{AG}(b \supset \mathbf{P}a)$
b **exceptif** a	*b* can occur iff *b* and *a* have never occurred.	$\mathbf{AG}(\neg\mathbf{P}a \wedge \mathbf{H}(\neg b) \equiv \mathbf{E}b)$
a **enables** b	*b* can occur iff *a* has already occurred but not *b*.	$\mathbf{AG}(\mathbf{P}a \wedge \mathbf{H}(\neg b) \equiv \mathbf{E}b)$
a **ensures** b	*b* will occur after *a* occurs, but *b* cannot occur without *a* having occurred.	$\mathbf{AG}(b \supset \mathbf{P}a \wedge a \supset \mathbf{F}b)$

The syntax and semantics of the logic supporting the specification of behavioural properties are currently being developed. In this logic, some properties of the underlying computational and interaction model will be fixed, such as:

- The initiation of an r&s interaction enables and ensures that a reply will be issued; we are working on an extension of the language that will provide primitives for assigning quality-of-service attributes such as the delay in which the reply is sent.
- A positive reply sets the pledge, which holds until the deadline, the commit or the deadline event occurs; the commit and the deadline events are enabled until either of them or the deadline occurs.
- Events occur only once during each "session", i.e. during each lifetime of an instance of a party.

We should point out that the style of specification that we adopt is quite different from recent proposals in the area of Semantic Web-Services (METEOR-S, OWL-S, SWSL, WSMF), which go little beyond a black-box, transformational approach based on concepts like pre- and post-conditions. These contribute to some extent towards a behavioural description of services but are confined to static/transformational aspects of black-box behaviour that only takes into account initial and final states of service execution. Therefore, they are not suitable for reasoning about conversational and stateful interactions as modelled in SRML-P. An exception is [15], which adopts an assumption/commitment style of specification as used for concurrent processes.

4 Components and Business Roles

In SRML-P, components instantiate *business roles*, which are specified by declaring a set of interactions and the way they are orchestrated. As an example, consider the business role of a supplier. A supplier can be involved in the following interactions:

```
INTERACTIONS
   r&s requestQuote
      🔔 which:product
      ✉ cost:money
   r&s orderGoods
      🔔 many:nat
      ✉ much:money
   s&r checkShipAvail
      🔔 which:product, many:nat
   rcv confirmShip
   rcv makePayment
   snd shipOrder
   ask how(product):money
   ask checkStock(product,nat):bool
   tll incStock(product,nat)
   tll decStock(product,nat)
```

Notice that the co-parties of the supplier in these interactions are not named; the specification models the business role played by the component independently of the way it is instantiated within any given system. Components are linked to their co-parties within modules through explicit wires as described in Section 6.

The way the declared interactions are orchestrated is specified through a set of variables that provide an abstract view of the state of the component, and a set of transitions that model the activities performed by the component, including the way it interacts with its co-parties.

A transition has an optional name and a number of possible features. For instance:

```
transition TQuote
   triggeredBy requestQuote🔔?
   guardedBy s=0
   effects which'=requestQuote.which
      ∧ much'=how(requestQuote.which)*1.2
      ∧ inStock'=false
      ∧ timeoutQuote'=false
      ∧ s'=1
   sends requestQuote✉!
      ∧ requestQuote.cost=much'
      ∧ requestQuote.Reply=true
      ∧ alertDate🔔!
      ∧ alertDate.Ref="quote"
      ∧ alertDate.Interval=7
```

- A trigger is a condition: typically, the occurrence of a receive-event.
- A guard is a condition that identifies the states in which the transition can take place – in *TQuote*, the state in which *s=0*. If the guard is false, a component that plays the specified role will not engage in the interaction.
- A sentence specifies the effects of the transition in the local state. We use *var'* to denote the value that a state variable *var* has after the transition. In the case above, we store business data and initialise the state variables *much*, *inStock* and *timeoutQuote*. Notice that, in the example, we use the synchronous interaction *how* to compute the cost that is going to be quoted. We will see that the co-party in this interaction is an external service that lists the current prices of goods.
- Another sentence specifies the events that are sent, including the values taken by their parameters. In this sentence, we use variables and primed variables as in the "effects"-section; the separation between the two sections is just logical and there are no dependencies between them. In the example, this consists in issuing the reply quoting the costs computed as mentioned and setting an *alertDate* with a 7-day interval – the period during which the quoted price is guaranteed.

Notice that, even if it is relatively easy to model a state machine in SRML-P, the way we model control flow is much more flexible because transitions are decoupled from interactions and changes to state variables. For instance, the transition *TAlert* can occur in any state after the request was issued:

```
transition TAlert
   triggeredBy alertDate✉?
   guardedBy
   effects alertDate.Ref="quote" ⊃ timeoutQuote'=true
      ∧ alert.Ref="goods" ∧ s=2 ⊃ s'=8
```

```
        sends alertDate.Ref="quote" ∧ s=1 ⊃ requestQuote💣!
           ∧ alert.Ref="goods" ∧ s=2 ⊃ orderGoods💣!
                ∧ incStock(which,many)
```

This transition is triggered when the supplier receives a notification from an *alertDate*; if the alert is concerned with the quote, it simply sets an internal timeout state variable so that the supplier knows how to calculate the costs of a subsequent order and it alerts its co-party that the timeout has occurred; if the alert is concerned with the goods and no commitment has been received, the supplier notifies its co-party and replenishes the local stock – *incStock(which,many)*. Notice that the latter is a synchronous interaction.

5 External Interfaces and Business Protocols

Besides components, a module in SRML-P may declare a number of (external) interfaces. These provide abstractions (types) of parties that can be interconnected with the components declared in the module either to provide or request services; this is what, in SCA, corresponds to "Entry Points" and "External Services".

External interfaces are specified through *business protocols*. Like orchestrations, protocols declare the interactions in which the external entities can be involved as parties. The difference is that, instead of an orchestration, we provide a set of properties that model the protocol that the co-party is expected to adhere to. For instance, the behaviour that a supplier expects from a warehouse is as follows:

```
BUSINESS PROTOCOL Warehouse is
___________________________________________________
   INTERACTIONS
      r&s check&lock
         🔔 which:product, many:nat
      snd confirm
   BEHAVIOUR
      check&lock🔔? exceptif true
      check&lock✉! ∧ check&lock.Reply ⊃
         alertDate🔔! ∧ alertDate.Interval=3 ∧
                 alertDate.Ref="goods"
      check&lock💣! ⊃ alertDate✉? ∧ alertDate.Ref="goods"
      check&lock⌛ ⊃ (check&lock✓? ensures confirm🔔!)
      check&lock✓? ⊃ (check&lock✝? exceptif confirm🔔!)
```

Notice that the interactions are again named from the point of view of the party concerned – the warehouse in the case at hand. The properties require the following:

- In the initial state the warehouse is ready to engage in *check&lock*.
- The deadline associated with *check&lock* is a timeout of 3 days with reference "goods" set when the reply is issued.
- A positive reply sets the pledge associated with *check&lock*, which ensures that *confirm* will be issued upon but not before receiving the commit.
- After the commit, *check&lock* can be revoked until *confirm* has been issued.

Protocols are also used for modelling the behaviour that users can expect from a service. This subsumes what, in [2], are called *external specifications*:

> *In particular, a trend that is gathering momentum is that of including, as part of the service description, not only the service interface, but also the business protocol supported by the service, i.e., the specification of which message exchange sequences are supported by the service, for example expressed in terms of constraints on the order in which service operations should be invoked.*

This is the case of customers:

```
BUSINESS PROTOCOL Customer is
  INTERACTIONS
     s&r howMuch
        🔔 which:product
        ✉ cost:money
     s&r buy
        🔔 many:nat
        ✉ much:money
     snd pay
     rcv ackShip
  BEHAVIOUR
     howMuch🔔? exceptif true
     howMuch✉? enables buy🔔!
     howMuch✉? ⊃ alertDate🔔! ∧ alertDate.Interval=7
        ∧ alertDate.Ref="quote"
     howMuch💣? ⊃ alertDate✉? ∧ alertDate.Ref="quote"
     howMuch✉? ⊃ howMuch.Reply
     howMuch⌛ ⊃ (buy🔔! ensures
        (buy✉? ∧ buy.Reply ⊃ buy.much=buy.many*howMuch.much))
     buy✉? ∧ buy.Reply ⊃ alertDate🔔! ∧ alertDate.Interval=3
        ∧ alertDate.Ref="goods"
     buy💣? ⊃ alertDate✉? ∧ alertDate.Ref="goods"
     buy⌛ ⊃ (pay🔔! ensures ackShip🔔?)
     pay🔔! ≡ buy✓!
     buy✓! ⊃ buy✞! exceptif ackShip🔔?
```

The properties offer the following behaviour:

- A request for *howMuch* is enabled at the start.
- A request for *buy* will be accepted after and only after a reply to *howMuch*.
- The deadline associated with *howMuch* is a timeout of 7 days set when the reply is received.
- A reply to *howMuch* is always positive; the corresponding pledge ensures that the cost associated with a subsequent order placed before the deadline will be the quoted one.
- The deadline associated with *buy* is a timeout of 3 days. This is why the warehouse is being requested to provide the same timeout.
- The pledge associated with *buy* ensures that *ackShip* will be issued upon and never before payment is issued.
- Payment is a commit to *buy*.
- *buy* can be revoked until *ackShip* has been issued.

Notice again that components and external interfaces are independent entities in the sense that they do not name the co-parties involved in the interactions that they support. These entities become connected in modules through internal wires.

6 Wires and Interaction Protocols

A module consists of a number of components and external interfaces (provides/requires) wired to one another. Wires are labelled by connectors that coordinate the interactions in which the parties are jointly involved. In SRML-P, we model the interaction protocols involved in these connectors as separate, reusable entities.

Just like business roles and protocols, an interaction protocol is specified in terms of a number of interactions. The "semantics" of the protocol is provided through a collection of sentences that establish how the interactions are coordinated, which may include routing events and transforming sent data to the format expected by the receiver. As an example, consider the following protocol:

```
INTERACTION PROTOCOL Custom1 is
  INTERACTIONS
    ask $S_1$(product,nat):bool
    tll $S_2$(product,nat)
    tll $S_3$(product,nat)
    rpl $R_1$(product):nat
    prf $R_2$(product,nat)
  COORDINATION
    $S_1$(p,n) = $R_1$(p)≥n
    $S_2$(p,n) ⊃ $R_2$(p,$R_1$(p)+n)
    $R_1$(p)≥n ∧ $S_3$(p,n) ⊃ $R_2$(p,$R_1$(p)–n)
    $R_1$(p)<n ⊃ ¬$S_3$(p,n)
```

This protocol is used by the wire *SS* that connects *Supplier* and *Stock* as follows:

SP Supplier	◇—	**SS**	—◇	**LS** Stock
ask checkStock	S_1	Custom1	R_1	**rpl** get
tll incStock	S_2		R_2	**prf** set
tll decStock	S_3			

The name bindings thus declared establish the following protocol:

```
checkStock(p,n)=(get(p)≥n)
incStock(p,n) ⊃ set(p,get(p)+n)
get(p)≥n ∧ decStock(p,n) ⊃ set(p,get(p)–n)
get(p)<n ⊃ ¬decStock(p,n)
```

That is, the boolean value returned by *checkStock(p,n)* as invoked by the supplier is computed by the local stock by checking if the value returned by *get(p)* is greater or equal to *n*. Notice that these are synchronous interactions. The protocol also stipulates that to a request from the supplier for *incStock(p,n)* the local stock executes *set(p,get(p)+n)*. Likewise, to a request from the supplier for *decStock(p,n)* the local stock executes *set(p,get(p)–n)* only if *get(p)* returns a value greater than or equal to *n*; otherwise, the request is not accepted.

The names used in interaction protocols are generic to facilitate reuse. In fact, families of protocols may be defined by parameterising the specification with the data sorts involved in the interactions. For instance, the following protocol is used between *Supplier* and *Customer*:

```
INTERACTION PROTOCOL Straight.I(d1)O(d2) is
   INTERACTIONS
      s&r S1
         🔔 i1:d1
         ✉ o1:d2
      r&s R1
         🔔 i1:d1
         ✉ o1:d2
   COORDINATION
      S1 ≡ R1
      S1.i1=R1.i1
      S1.o1=R1.o1
```

This is a "standard" protocol that connects directly two entities over two interactions with one 🔔– and one ✉-parameter. This protocol is used twice in the following wire to connect different interactions between *Supplier* and *Customer:*

SP Supplier	◇—	**CS**	—◇	**CR** Customer
r&s requestQuote 🔔 which ✉ cost	R_1 i_1 o_1	Straight	S_1 i_1 o_1	**s&r** howMuch 🔔 which ✉ cost
r&s orderGoods 🔔 which ✉ cost	R_1 i_1 o_1	Straight	S_1 i_1 o_1	**s&r** buy 🔔 which ✉ cost
rcv makePayment	R_1	Straight	S_1	**snd** pay
snd shipOrder	S_1	Straight	R_1	**rcv** ackShip

The other protocol used in this wire is an even simpler version involves no parameters:

```
INTERACTION PROTOCOL Straight is
   INTERACTIONS
      snd S1
      rcv R1
   COORDINATION
      S1 ≡ R1
```

The name bindings establish straightforward connections such as:

```
howMuch ≡ requestQuote
howMuch.which = requestQuote.which
howMuch.cost = requestQuote.cost
buy ≡ orderGoods
buy.which = orderGoods.which
buy.much = orderGoods.much
pay ≡ makePayment
ackShip ≡ shipOrder
```

Interaction protocols are considered as first-class objects because we want to use them to assign properties to wires that reflect constraints on the underlying run-time environment. These may concern data transmission, synchronous/asynchronous connectivity, distribution, and other non-functional properties such as security.

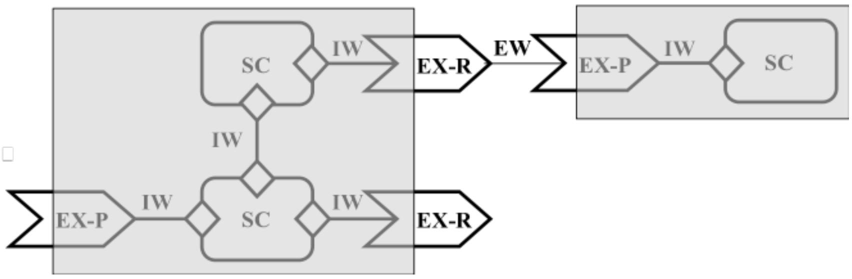
7 Concluding Remarks and Further Work

In this paper, we have described some of the primitives that are being proposed for the SENSORIA Reference Modelling Language in order to support building systems in service-oriented architectures using "technology agnostic" terms. More specifically, we have focused on the language that supports the underlying composition model. This is a minimalist language that follows a recent proposal for a Service Component Architecture [14] that "builds on emerging best practices of removing or abstracting middleware programming model dependencies from business logic". However, whereas the SCA-consortium concentrates on the definition of an open specification that supports a variety of component implementation and interface types, and on the deployment, administration and configuration of SCA-based applications, our goal is to development a mathematical framework in which service-modelling primitives can be formally defined and application models can be reasoned about.

This is why we are developing a logic for specifying and reasoning about interactions in the conversational mode that characterises services. The primitives that we are proposing take into account proposals that have been made for Web-Service Conversation [4], in other modelling languages such as ORC [13], and in calculi such as Sagas [5]; they take into account that interactions are stateful and provide first-class notions such as reply, commit, compensation and pledge.

The core of our paper focused on the notion of module, which we adapted from SCA. Modules in SRML-P are the basic units of composition. They include external interfaces for required and provided services, and a number of components whose orchestrations ensure that the properties offered on the provides-external interfaces are guaranteed by the connections established by the wires assuming that the services requested satisfy the properties declared on the requires-external interfaces. An algebraic formalisation of this notion of module can be found in [9], which includes the correctness condition. We have also added a notion of parameter through which we can configure chosen aspects of a module such as timeouts; such parameters can be instantiated at run-time as part of a negotiation process.

Modules can be assembled together to make complex systems in a way that is similar to SCA, i.e. by linking requires-external interfaces of a module with provides-external interfaces of other modules via external wires. External wires carry a proof-obligation to ensure that the properties offered by the provides-interface imply those declared by the requires-interface.



An assembly of modules defining a SRML-P system; EW–external wire

SRML-P also supports a way of offering a system as a module, i.e. of turning an assembly of services into a composite service that can be published and discovered on its own. This can be useful, for instance, when one wants to put together a number of services that, individually, offer only partial matches for a given required external interface but, in a suitable configuration, can provide a suitable match. The operation that collapses a system into a module internalises the external wires and forgets the external specifications. An algebraic semantics of module interconnection and composition can be found in [9] based on categorical constructions similar to those used in algebraic specification [7] and software architecture [10].

Finally, we are also developing a notion of configuration for SRML-P. A configuration is a collection of components wired together that models a run-time composition of service components. A configuration results from having one or more clients using the services provided by a given module, possibly resulting from a complex system, with no external interfaces, i.e. with all required external interfaces wired-in. It is at the level of configurations that we address run-time aspects of service composition such as sessions, as well as notions of persistence. Research is under way to provide primitives for managing configurations with a semantics based on graph-transformations [7], as used, for instance, in [3,16].

Acknowledgments

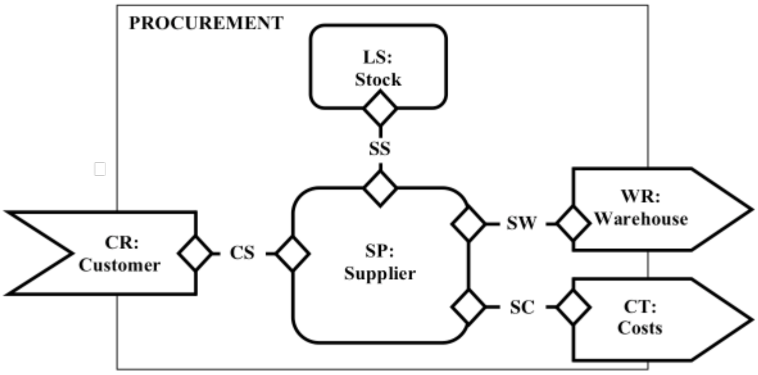
J. Fiadeiro was partially supported by a grant from the Royal Society (UK) and A. Lopes by the Foundation for Science and Technology (Portugal) during an extended stay at the University of Pisa during April and May 2006. We wish to thank our hosts for the facilities and opportunities for discussion. We would like to thank Luís Andrade, Roberto Bruni, Rocco de Nicola, Giorgios Koutsoukos, Ugo Montanari and Martin Wirsing for their comments on previous versions of this paper.

References

1. G. Alonso, F. Casati, H. Kuno, V. Machiraju (2004) *Web Services*. Springer, Berlin Heidelberg New York
2. K. Baïna, B. Benatallah, F. Casati, F. Toumani (2004) Model-driven web service development. In A. Persson, J. Stirna (eds): *CAiSE'05. LNCS, vol 3084*. Springer, Berlin Heidelberg New York, pp 290–306
3. L. Baresi, R. Heckel, S, Thöne, D. Varró (2003) Modeling and validation of service-oriented architectures: Application vs style. In A. Persson, J. Stirna (eds): *ESEC'03. LNCS, vol 3084*. Springer, Berlin Heidelberg New York, pp 290–306
4. B. Benatallah, F. Casati, F. Toumani (2004) Web service conversation modelling. *IEEE Internet Computing* 8(1):46–54
5. R. Bruni, H. Melgratti, U. Montanari (2005) Theoretical foundations for compensations in flow composition languages. In *POPL'05*. ACM Press, New York, pp 209-220
6. F. Curbera, R. Khalaf, N. Mukhi, S. Tai, S. Weerawarana (2003) The next step in web services. *CACM* 46(10):29–34
7. H. Ehrig, K. Ehrig, U. Prange, G. Taentzer (2006) *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs on Theoretical Computer Science. Springer, Berlin Heidelberg New York
8. H. Ehrig, B. Mahr (2005) *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. EATCS Monographs on Theoretical Computer Science, vol 21. Springer, Berlin Heidelberg New York
9. J. L. Fiadeiro, A. Lopes, L. Bocchi (2006) *The SENSORIA Reference Modelling Language: Primitives for Service Description*. Available from www.sensoria-ist.eu
10. J. L. Fiadeiro, A. Lopes, M. Wermelinger (2003) A mathematical semantics for architectural connectors. In: R. Backhouse, J. Gibbons (eds) *Generic Programming. LNCS, vol 2793*. Springer, Berlin Heidelberg New York, pp 190–234
11. I. Foster, C. Kesselman (eds) (2004) *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA
12. R. Goldblatt (1987) *Logics of Time and Computation*. CSLI, Stanford
13. J. Misra, W. Cook (2006) Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modelling*. To appear
14. SCA Consortium (2005) *Building Systems using a Service Oriented Architecture*. Whitepaper available from www-128.ibm.com/developerworks/library/specification/ws-sca/
15. M. Solanki, A. Cau and H. Zedan (2004) Augmenting semantic web service description with compositional specification. In *WWW'04*. ACM Press, New York, pp 544–552
16. M. Wermelinger, A. Lopes, J. L. Fiadeiro (2001) A graph-based architectural (re)-configuration language. In V.Gruhn (ed): *ESEC/FSE'01*. ACM Press, New York, pp 21–32

Appendix – The procurement service

In this appendix, we model the procurement business process used in the paper, involving a supplier, a warehouse, a local stock, and a price look-up facility.



PROCUREMENT consists of:

- CR – the external interface of the service provided by the module, of type *Customer*;
- WR – the external interface of a service required for shipping the product if it is not available locally, of type *Warehouse*;
- CT – the external interface of a service required for quoting the current market costs of products, of type *Costs*;
- SP – a component that coordinates the business process, of type *Supplier*;
- LS – a component that provides local storage of products, of type *Stock*
- *CS*, *SS, SW*, *SC* – four internal wires that make explicit the partner relationship between *CR* and *SP*, *SP* and *LS*, *SP* and *WR*, and *SP* and *CT,* respectively.

The components, external interfaces and protocols required for the definition of *PROCUREMENT* are collected at the end of the appendix.

```
MODULE Procurement is

   COMPONENTS
      SP: Supplier
      LS: Stock

   PROVIDES
      CR: Customer

   REQUIRES
      WR: Warehouse
      CT: Costs
```

WIRES

SP Supplier	◇—	**SS**	—◇	**LS** Stock
ask checkStock	S_1	Custom1	R_1	**rpl** get
tll incStock	S_2		R_2	**prf** set
tll decStock	S_3			

SP Supplier	◇—	**SC**	—◇	**CT** Costs
ask how	S_1	AskTll	R_1	**tll** much

SP Supplier	◇—	**SW**	—◇	**WH** Warehouse
s&r checkShipAvail	S_1	Straight	R_1	**r&s** check&lock
🔔 which	i_1		i_1	🔔 which
many	i_2		i_2	many
rcv confirmShip	R_1	Straight	S_1	**snd** confirm

SP Supplier	◇—	**CS**	—◇	**CR** Customer
r&s requestQuote	R_1	Straight	S_1	**s&r** howMuch
🔔 which	i_1		i_1	🔔 which
✉ cost	o_1		o_1	✉ cost
r&s orderGoods	R_1	Straight	S_1	**s&r** buy
🔔 which	i_1		i_1	🔔 which
✉ cost	o_1		o_1	✉ cost
rcv makePayment	R_1	Straight	S_1	**snd** pay
snd shipOrder	S_1	Straight	R_1	**rcv** ackShip

END MODULE

SPECIFICATIONS

BUSINESS ROLE Stock **is**

```
INTERACTIONS
   rpl get(product):nat
   prf set(product,nat)

ORCHESTRATION
   local qoh:product→nat
   transition
      triggeredBy get(p)
      sends qoh(p)
   transition
      triggeredBy set(p,n)
      effects qoh(p)'=n
```

```
BUSINESS ROLE Supplier is
   INTERACTIONS
      r&s requestQuote
         🔔 which:product
         ✉ cost:money
      r&s orderGoods
         🔔 many:nat
         ✉ much:money
      rcv makePayment
      snd shipOrder
      s&r checkShipAvail
         🔔 which:product, many:nat
      rcv confirmShip
      ask how(product):money
      ask checkStock(product,nat):bool
      tll incStock(product,nat)
      tll decStock(product,nat)

   ORCHESTRATION
      local s:[0..8], inStock:bool, which:product, many:nat,
                 much:money, timeoutQuote:bool
      initialisation
         s=0
      termination
         s=8
      transition TQuote
         triggeredBy requestQuote🔔?
         guardedBy s=0
         effects which′=requestQuote.which
            ∧ much′=how(requestQuote.which)*1.2
            ∧ inStock′=false
            ∧ timeoutQuote′=false
            ∧ s′=1
         sends requestQuote✉!
            ∧ requestQuote.cost=much′
            ∧ requestQuote.Reply=true
            ∧ alertDate🔔!
            ∧ alertDate.Ref=”quote”
            ∧ alertDate.Interval=7

      transition TAlert
         triggeredBy alertDate✉?
         guardedBy
         effects alert.Ref=”quote” ∧ s=1 ⊃ timeoutQuote′=true
            ∧ alert.Ref=“goods” ∧ s=2 ⊃ s′=8
         sends alert.Ref=”quote” ∧ s=1 ⊃ requestQuote💣!
            ∧ alert.Ref=“goods” ∧ s=2 ⊃ orderGoods💣!
                 ∧ incStock(which,many)

      transition TimeoutOrder
         triggeredBy checkShipAvail💣?
         guardedBy
         effects s=4 ⊃ s′=8
         sends s=4 ⊃ orderGoods💣!
```

```
transition TOrder
  triggeredBy orderGoods🔔?
  guardedBy s=1
  effects many′=orderGoods🔔.many
     ∧ timeoutQuote ⊃
        much′=orderGoods.many*how(requestQuote.which)*1.2
     ∧ ¬timeoutQuote ⊃ much′=orderGoods.many*much
     ∧ checkStock(which,orderGoods.many) ⊃ s′=2
              ∧ inStock′=true
     ∧ ¬checkStock(which,orderGoods.many) ⊃ s′=3
              ∧ inStock′=false
  sends inStock′ ⊃ decStock(which,many)
              ∧ orderGoods✉!
              ∧ orderGoods.much=much′
              ∧ orderGoods.Reply=true
              ∧ alertDate🔔!
              ∧ alertDate.Ref=”goods”
              ∧ alertDate.Interval=3
     ∧ ¬inStock′ ⊃ checkShipAvail🔔!
              ∧ checkShipAvail.which=which
              ∧ checkShipAvail.many=many′

transition TWare
  triggeredBy checkShipAvail✉?
  guardedBy s=3
  effects checkShipAvail.Reply ⊃ s′=4
     ∧ ¬checkShipAvail.Reply ⊃ s′=8
  sends checkShipAvail.Reply ⊃ orderGoods✉!
              ∧ orderGoods.Reply=true
              ∧ orderGoods.much=much
     ∧ ¬checkShipAvail.Reply ⊃ orderGoods✉!
              ∧ orderGoods.Reply=false

transition TPay
  triggeredBy makePayment🔔?
  guardedBy (s=2 ∨ s=4)
  effects s=2 ⊃ s′=5
     ∧ s=4 ⊃ s′=6
  sends s=4 ⊃ checkShipAvail✓!

transition TConfirm
  triggeredBy confirmShip🔔?
  guardedBy s=6
  effects s′=7

transition TShip
  triggeredBy
  guardedBy s=5 ∨ s=7
  effects s′=8
  sends shipOrder🔔!

transition TAbort
  triggeredBy orderGoods✝?
  guardedBy (s=5 ∨ s=6)
  effects s′=8
  sends s=5 ⊃ incStock(which,many)
     ∧ s=6 ⊃ checkShipAvail✝!
```

```
BUSINESS PROTOCOL Warehouse is
  INTERACTIONS
    r&s check&lock
      🔔 which:product, many:nat
    snd confirm
  BEHAVIOUR
    check&lock🔔? exceptif true
    check&lock⌧! ∧ check&lock.Reply ⊃
      alertDate🔔! ∧ alertDate.Interval=3 ∧
              alertDate.Ref="goods"
    check&lock💣! ⊃ alertDate⌧? ∧ alertDate.Ref="goods"
    check&lock⧗ ⊃ (check&lock✓? ensures confirm🔔!)
    check&lock✓? ⊃ (check&lock✞? exceptif confirm🔔!)
```

```
BUSINESS PROTOCOL Costs is
  INTERACTIONS
    rpl much(product):money
```

```
BUSINESS PROTOCOL Customer is
  INTERACTIONS
    s&r howMuch
      🔔 which:product
      ⌧ cost:money
    s&r buy
      🔔 many:nat
      ⌧ much:money
    snd pay
    rcv ackShip
  BEHAVIOUR
    howMuch🔔? exceptif true
    howMuch⌧? enables buy🔔!
    howMuch⌧? ⊃ alertDate🔔! ∧ alertDate.Interval=7
      ∧ alertDate.Ref="quote"
    howMuch💣? ⊃ alertDate⌧? ∧ alertDate.Ref="quote"
    howMuch⌧? ⊃ howMuch.Reply
    howMuch⧗ ⊃ (buy🔔! ensures
      (buy⌧? ∧ buy.Reply ⊃ buy.much=buy.many*howMuch.much))
    buy⌧? ∧ buy.Reply ⊃ alertDate🔔! ∧ alertDate.Interval=3
      ∧ alertDate.Ref="goods"
    buy💣? ⊃ alertDate⌧? ∧ alertDate.Ref="goods"
    buy⧗ ⊃ (pay🔔! ensures ackShip🔔?)
    pay🔔! ≡ buy✓!
    buy✓! ⊃ buy✞! exceptif ackShip🔔?
```

INTERACTION PROTOCOL Straight **is**

INTERACTIONS
snd S_1
rcv R_1
COORDINATION
$S_1 \equiv R_1$

INTERACTION PROTOCOL Straight.I(d_1)O(d_2) **is**

INTERACTIONS
s&r S_1
🔔 i_1:d_1
✉ o_1:d_2
r&s R_1
🔔 i_1:d_1
✉ o_1:d_2
COORDINATION
$S_1 \equiv R_1$
$S_1.i_1=R_1.i_1$
$S_1.o_1=R_1.o_1$

INTERACTION PROTOCOL Straight.I(d_1,d_2) **is**

INTERACTIONS
s&r S_1
🔔 i_1:d_1, i_2:d_2
r&s R_1
🔔 i_1:d_1, i_2:d_2
COORDINATION
$S_1 \equiv R_1$
$S_1.i_1=R_1.i_1$
$S_1.i_2=R_1.i_2$

INTERACTION PROTOCOL Custom1 **is**

INTERACTIONS
ask S_1(product,nat):bool
tll S_2(product,nat)
tll S_3(product,nat)
rpl R_1(product):nat
prf R_2(product,nat)
COORDINATION
$S_1(p,n) = R_1(p)\geq n$
$S_2(p,n) \supset R_2(p,R_1(p)+n)$
$R_1(p)\geq n \wedge S_3(p,n) \supset R_2(p,R_1(p)-n)$
$R_1(p)<n \supset \neg S_3(p,n)$

INTERACTION PROTOCOL AskTll(d_1,d_2) **is**

INTERACTIONS
ask $S_1(d_1)$:d_2
tll $R_1(d_1)$:d_2
COORDINATION
$S_1(x) = R_1(x)$