A Coordination Model for Service-Oriented Interactions[*]

João Abreu and José Luiz Fiadeiro

Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
{jpad2,jose}@mcs.le.ac.uk

Abstract. We present a formal model for the coordination of interactions in service-oriented systems. This model provides a declarative semantics for the language SRML that is being developed under the FET-GC2 project SENSORIA for modelling and reasoning about complex services at the abstract business level. In SRML, interactions are conversational in the sense that they involve a number of correlated events that capture phenomena that are typical of SOC like committing to a pledge or revoking the effects of a deal. Events are exchanged across wires that connect the parties involved in the provision of the service.

1 Introduction

One of the challenges raised by service-oriented computing (SOC) is to develop a semantic model that is rich enough for capturing the new kinds of interactions that it introduces but also abstract enough to support the modelling of systems at the "business level", i.e. independently of the middleware programming model. It is fair to say that the bulk of the research that is being published in this area is directed to the languages and infrastructures that support Web Services [2], which is understandable because this is the area where industry has its most immediate interests. Our research is being developed within a FET (Future Emerging Technologies) project — SENSORIA [17] — so as to provide foundations for SOC as a paradigm and not just a technology.

In particular, we have been developing a reference modelling language (SRML) through which we would like to support building systems with service-oriented architectures in "technology agnostic" terms. SRML is based on a semantic model (discussed in this paper) that provides a layer of abstraction above the languages in which services are programmed and the middleware that supports the coordination of interactions [2]. In [7] we have shown that SRML is expressive enough to accommodate orchestrations programmed in languages such as BPEL. In previous papers we have provided an overview of the SRML language [10] and of the algebraic semantics of service composition [11]. In this paper, we present

[*] This work was partially sponsored through the IST-2005-16004 Integrated Project SENSORIA: Software Engineering for Service-Oriented Overlay Computers.

a formal model for the primitives that we are using for the coordination of interactions in service-oriented systems.

SRML supports three different levels of "coordination" in SOC. One concerns the process of discovery of external services that may be required for a certain computation. In SRML, this process is not programmed as part of the computational process performed by services but handled separately; one of the novelties of SOC is precisely in the externalisation of discovery — see [6] for more details about the discovery and binding of new services in SRML. Another level concerns the coordination (orchestration) of the various parties that, together, deliver a complex service. In SRML, we adopt a "classical" architectural approach in which this type of coordination is performed by connectors (in the sense of REO [3]) that link together the different parties involved in the delivery of the service. Other approaches adopt workflow models [16]. We have discussed this level of coordination in [1] and, although briefly discussed in Section 2, it is not the core of our paper.

Our main contribution in this paper is at the third level of coordination: the one that needs to be established between the different events that are involved in interactions. In our model, interactions are conversational in the sense that they involve a number of correlated events between two parties. To the best of our knowledge, this is the first formal model proposed for SOC that adopts a rich ontology of interactions.

In section 2 we give an overview of the SRML approach to the specification of service-oriented architectures and the intuitive semantics that is associated with it; we illustrate it with examples taken from the specification of a travel booking service. In section 3 we formalize the notions presented in section 2 by defining our model of service-oriented architectures and computation, over which SRML specifications should be interpreted. Finally, section 4 concludes and outlines further work already being carried out.

2 Modelling Complex Services in SRML

2.1 The Compositional Model

Our approach to service-oriented specification follows recent proposals by the Service Component Architecture (SCA) initiative — for a deeper discussion on the relation between SRML and SCA refer to [10]. Like in SCA, the architectural unit for specifying a complex service in SRML is the module. Modules specify how a set of independent parties are interconnected and interact to provide the behaviour of the service. A module consists of an architecture, i.e. the definition of which pairs of parties are connected through wires, and a specification for each of the parties and each of the wires. Figure 1 shows the structure of the module $TravelBooking$, which models a service that manages the booking of a flight and a hotel.

The service is assembled by connecting an internal component $BookingAgent$ to the external services $PayAgent$, $HotelAgent$ and $FlightAgent$ and the persistent component (a database of users) $UsrDB$. The difference between the three

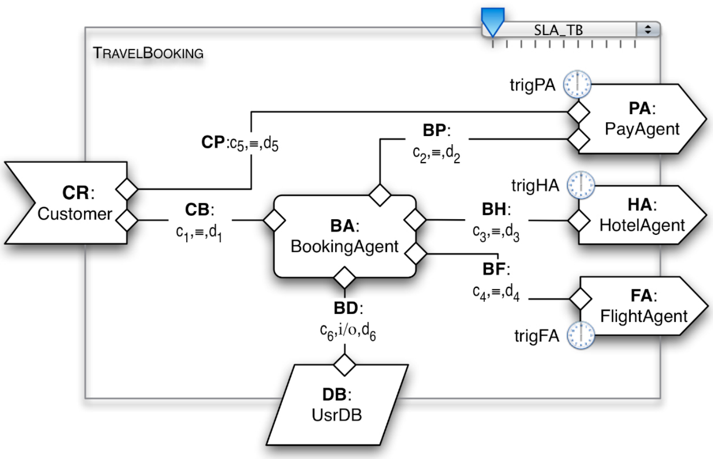

Fig. 1. The structure of the module *TravelBooking*

kinds of entities is intrinsic to SOC: internal components are created each time the service is invoked and killed when the service terminates; external services are procured and bound to the other parties at run time; persistent components are part of the business environment in which the service operates — they are not created nor destroyed by the service, and they are not discovered but directly invoked as in component-based systems. *Customer* is the interface through which service requesters interact with the *TravelBooking* service. In SRML, interactions are peer-to-peer between pairs of entities connected through wires — *CB*, *CP*, *BP*, *BH*, *BF* and *BD* are the wires in *TravelBooking*. Complex services like *TravelBooking* establish multi-party collaborations by orchestrating their interactions.

The specification of each of the parties contains a declaration of the interactions the party can be involved in and a specification of the properties that can be observed of these interactions during a session. If the party is an internal component of the service, this specification is an orchestration given in terms of state transitions — the language of business roles. If the party is the interface of an external service or persistent component, the specification consists of a set of temporal properties expressed in temporal logic — the language of business protocols. Figure 2 shows part of the specification of the component *BookingAgent* - the orchestration resorts to a set of locally declared variables in order to define the state transitions the component is involved in. Figure 3 shows the specification of the business protocol that the hotel agent service is expected to engage in — the language involves abbreviations of temporal logic formulae. The use of temporal logic has also been adopted by workflow-based approaches to SOC; in [16] constraint templates based on linear temporal logic are used to capture common specification patterns for service flows. In order to capture patterns of service-oriented interactions we use abbreviations of an action/state branching time logic based on UCTL [12]. This new logic is being developed within SENSORIA together with our partners at *ISTI-CNR (Pisa)*. Details about this logic and on how it encodes the patterns of service-oriented interactions used in SRML specifications will be presented in forthcoming publications.

The specification of each wire consists of a set of *connectors* [1] that are responsible for binding and coordinating, through interaction protocols, the complex interactions that are declared locally in the specifications of the two parties that the wire connects (much in the sense of [14]). Figure 4 shows the specification of the wire BH that connects $BookingAgent$ to $HotelAgent$. The only interaction that exist between these two parties is named $bookHotel$ from the point of view $BookingAgent$ and is named $lockHotel$ from the point of view of $HotelAgent$. The reason that interactions can be named differently in the two parties is precisely due to the fact that complex services are put together at run time without a-priori knowledge of the parties that will be involved. Because of this, we need to rely on the interaction protocols of the wires to establish how these interactions are related and coordinated. In this paper, we will not discuss interaction protocols in any great length; see [1] instead. This is because such connector-based coordination is by now well understood. The contribution of this paper is in the coordination model that we propose for the different events that occur as part of the interactions. The following sections will clarify the examples shown in figures 2, 3 and 4 — in particular, the meaning of the icons and symbols that are used will be explained.

2.2 Service-Oriented Interactions and Events

In service-oriented systems, typical interactions are of a conversational type and cannot be modelled as simple state transitions because they involve a durative asynchronous exchange of correlated events. In SRML, two-way interactions capture a pattern of dialogue that is prevalent in service-oriented systems: a party sends a request to a co-party that replies either positively by making a pledge to deliver a set of properties (i.e. it gives some kind of guarantee) or negatively, in which case the interaction ends; if the answer is positive the party that made the request can commit by accepting the pledge or refuse the pledge and cancel the interaction. If and after the requester commits, a revoke may be available that compensates for the effects of the pledge. One-way interactions are also supported in SRML: they capture situations in which a party sends a single event and does not expect a reply from the co-party. This type of interaction has only this one event associated with it. The set of events associated with an interaction a is shown in the following table:

a🔔	The initiation-event of a.
a🖂	The reply-event of a.
$a\checkmark$	The commit-event of a.
a✗	The cancel-event of a.
a✞	The revoke-event of a.

```
BUSINESS ROLE BookingAgent is

  INTERACTIONS
    r&s bookTrip
      🔔 from,to:airport; out,in:date
      ✉ fconf:fcode; hconf:hcode; amount:moneyvalue
    s&r bookFlight
      🔔 from,to:airport; out,in:date; traveller:usrdata
      ✉ fconf:fcode; amount:moneyvalue;
        beneficiary:accountn; payService:serviceId
    s&r payment
      🔔 amount:moneyvalue; beneficiary:accountn
        originator:usrdata; cardNo:paydata
      ✉ proof:pcode
    s&r bookHotel
      🔔 checkin,checkout:date,
        traveller:usrdata
      ✉ hconf:hcode
    …

  ORCHESTRATION
    local
      s:[START, LOGGED, QUERIED, FLIGHT_OK, HOTEL_OK,
        CONFIRMED, END_PAYED, END_UNBOOKED, COMPENSATING,
        END_COMPENSATED]; login:Boolean;
        traveller:usrdata; travcard:paydata
    transition Request
     | triggeredBy bookTrip🔔?
     | guardedBy s=LOGGED
     | effects bookTrip🔔.out>today ⊃ s'=QUERIED
     |    ∧ bookTrip🔔.out≤today ⊃ s'=END_UNBOOKED
     | sends bookTrip🔔.out>today ⊃ bookFlight🔔!
     |        ∧ bookFlight🔔.from=bookTrip🔔.from
     |        ∧ bookFlight🔔.to=bookTrip🔔.to
     |        ∧ bookFlight🔔.out=bookTrip🔔.out
     |        ∧ bookFlight🔔.in=bookTrip🔔.in
     |        ∧ bookFlight🔔.traveller=traveller
     |    ∧ bookTrip🔔.out≤today ⊃ bookTrip✉!
     |        ∧ bookTrip✉.Reply=False
    transition TripCommit
     | triggeredBy bookTrip✓?
     | guardedBy s=HOTEL_OK
     | effects s'=CONFIRMED
     | sends bookFlight✓! ∧ bookHotel✓!∧ payment🔔!
     |        ∧ payment🔔.amount=bookFlight✉.amount
     |        ∧ payment🔔.beneficiary=
     |            bookFlight✉.beneficiary
     |        ∧ payment🔔.originator=traveller
     |        ∧ payment🔔.cardNo=travcard
```

Fig. 2. An extract from the specification of the component *BookingAgent* written in the language of business roles. Some of the interactions in which *BookingAgent* is involved in — *bookTrip*, *bookFlight*, *payment* and *bookHotel* — are declared. A set of local state variables is also declared and the specifications of transitions *Request* and *TripCommit* are shown.

Associated with every positive reply there is a deadline, *a.useBy*, for the party to reply within which the co-party offers a pledge. After the deadline is over there is no guarantee that the co-party will interact with the party any longer. Figure 5 represents the intuitive semantics of a two-way interaction when the co-party

```
BUSINESS PROTOCOL HotelAgent is
  INTERACTIONS
     r&s lockHotel
        🔔 checkin,checkout:date; name:usrdata
        ✉ hconf:hcode
  BEHAVIOUR
     initiallyEnabled lockHotel🔔?
     lockHotel✓? enables lockHotel✝? until
         today < lockHotel🔔.checkin
```

Fig. 3. The specification of the external interface *HotelAgent* written in the language of business protocols. *HotelAgent* is involved in one interaction named *lockHotel* that models the booking of a room in a hotel. Some properties of this interaction are specified: a room booking can be initiated once the service is instantiated and a room reservation can be canceled up until the check-in date.

BA BookingAgent	c_3	**BH**	d_3	**HA** HotelAgent
s&r bookHotel	S		R	**r&s** lockHotel
🔔 checkin	i_1		i_1	🔔 checkin
checkout	i_2	≡	i_2	checkout
traveller	i_3		i_3	name
✉ hconf	o_1		o_1	✉ hconf

Fig. 4. The specification of the wire *BH* that connects *BookingAgent* to *HotelAgent*. ≡ denotes a straight interaction protocol [1] that binds interaction *bookHotel* (declared in the specifications of *BookingAgent*) to interaction *lockHotel* (declared in the specification of *HotelAgent*).

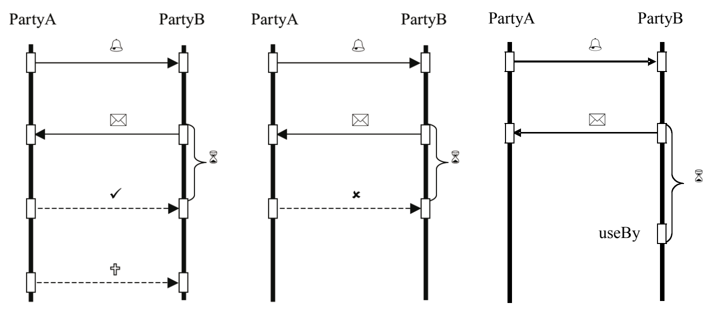

Fig. 5. The intuitive semantics of two-way interactions

replies positively. In the case on the left, the initiator commits to the pledge; a revoke may occur later on, compensating the effects of the commit-event. In the middle, there is a cancellation; in this situation, a revoke is not available. In the case on the right, the deadline occurs without a commit or cancel having occurred.

In specifications one-way interactions are typed by either *snd* or *rcv* to distinguish between the points of view of the sending party and the receiving co-party, respectively. The equivalent types for two-way interactions are *s&r* (send and receive) and *r&s* (receive and send). For instance, the specification of HotelAgent, shown in figure 3, declares a two-way interaction lockHotel typed with *r&s* to mean that the co-party that engages with hotel agent in this interaction is responsible for initiating it by requesting a hotel booking.

2.3 Asynchronous Coordination

Parties engage in interactions independently of their co-parties, i.e. the workflow that determines when a party interacts, by publishing an event or processing it, is independent of the way these events are transmited [1]. Wires are responsible for establishing and coordinating interactions between parties; events are carried from one party to the other by the wire that connects them. Associated with each wire there is a delay that represents the maximum time that the wire takes to deliver each event to the receiving party after it is sent. The delay of each wire is set at run time as part of the service level agreement that is negotiated when external services need to be procured [6].

We use $e!$ to refer to the publishing of event e and $e?$ to refer to its processing. In the specification of component *BookingAgent*, shown in figure 2, there is a transition named *TripCommit* that is triggered by the processing of event *bookTrip*✓. The effect of this transition is that of publishing events *bookFlight*✓, *bookHotel*✓ and *payment*🔔.

It is also important to distinguish between the notion of processing an event and that of executing it. Parties are not always in a state in which they are ready to engage in a given interaction. For instance, in order for the processing of event *bookTrip*✓ to have the effect described in transition *TripCommit*, shown in figure 2, the *BookingAgent* needs to be in a state in which the local variable s is set to $HOTEL\_OK$; we say that *bookTrip*✓ is enabled in such states. If the event is processed in a state in which *BookingAgent* is not ready to execute it, then the event is discarded. In the case of interaction *lockHotel* in *HotelAgent*, shown in figure 3, the revoke-event *lockHotel*✞ becomes enabled by the execution of the commit event *lockHotel*✓ that confirms a reservation, but it is enabled only before the check-in - this is specified through the second property of *HotelAgent*.

3 The Semantic Model Underlying SRML

In this section we formalize the notions that were given informally in section 2. Throughout the rest of the paper we assume a fixed data signature $\Sigma =$

$\langle D, F\rangle$, where D is a set of sorts and F is a $D^* \times D$-indexed family of sets of operations. We further assume that $time, boolean \in D$ are sorts that represent the usual concepts of time and truth values. We also assume a fixed algebra $\mathcal{U}$ for interpreting Σ.

3.1 Signatures

We use the notion of signature to characterize a service-oriented architecture and as the basis for defining the models of behavior that are valid for that architecture.

Definition 1 (SRML Interaction Signature)
A SRML interaction signature (signature for short) is a tuple $\langle COMP, WIRE, 2WAY, 1WAY\rangle$ *where:*

- $\langle COMP, WIRE\rangle$ *is a simple graph (undirected, without self-loops or multiple edges) where* $COMP$ *is the set of nodes (the parties that form the service) and* $WIRE$ *is the set of edges (the wires that connect the parties).*
- $2WAY$ *and* $1WAY$ *are* $COMP \times COMP$*-indexed families of mutually disjoint sets of names of asynchronous two-way and one-way interactions, respectively, each taking place between a pair of parties; we use* INT *to refer to* $2WAY \cup 1WAY$.
- *For every* $c, c' \in COMP$, $INT_{\langle c,c'\rangle} = \emptyset$ *if* $\langle c, c'\rangle \notin WIRE$, *i.e. there are no interactions between components that are not connected by a wire.*

The graph $\langle COMP, WIRE\rangle$ defines the set of parties that compose the service and how they are interconnected by wires. The graph does not have multiple edges, meaning that for every two parties there is either a single wire connecting them or they are not directly connected. Also the graph does not have loops, meaning that a party cannot be connected to itself. The graph is undirected because wires do not have a direction associated with them; wires are able to transmit events both ways. Interactions are directed: if interaction i belongs to $INT_{\langle c,c'\rangle}$ this means that the interaction is initiated by party c. Obviously, in this case, there needs to be a wire between parties c and c' for the interaction to take place; this is captured by the last condition of the definition.

Throughout the rest of the paper we will consider a fixed signature $S = \langle COMP, WIRE, 2WAY, 1WAY\rangle$ over which all definitions will be given.

3.2 Events and Pledges

A signature defines which interactions are established between the parties of the system. This information allows us to formalize the notion of event that was introduced in 2.2. We do this by defining which events can be sent and received by each of the parties.

Definition 2 (Events)
For every $a \in INT$ *and* $x \in COMP$, *the set* $E_x(a)$ *of events associated with interaction* a *that are received by a party* x *is defined as follows:*

If $a \in 2WAY_{\langle c,c' \rangle}$ *then*

$E_c(a) = \{a\text{✉}\}$

$E_{c'}(a) = \{a\text{🔔}, a\checkmark, a\text{✗}, a\text{✞}\}$

$E_{c''}(a) = \emptyset$ *for any other* $c'' \in COMP$

If $a \in 1WAY_{\langle c,c' \rangle}$ *then*

$E_c(a) = \emptyset$

$E_{c'}(a) = \{a\text{🔔}\}$

$E_{c''}(a) = \emptyset$ *for any other* $c'' \in COMP$

We also define the following sets:

- $E_c = \bigcup\{E_c(a) : a \in INT\}$ *is the set of all events that can be received by party* c*.*
- $E(a) = E_c(a) \cup E_{c'}(a)$ *where* $a \in INT_{\langle c,c' \rangle}$ *is the set of events associated with interaction* a*.*
- $E_{\langle c,c' \rangle} = \bigcup\{E(a) : a \in INT_{\langle c,c' \rangle} \vee a \in INT_{\langle c',c \rangle}\}$ *is the set of all events that are carried by wire* $\langle c, c' \rangle$*.*
- $E = \bigcup\{E(a) : a \in INT\}$ *is the set of all events that can happen in the system.*

We see E *as a* $WIRE$*-indexed or a* $COMP$*-indexed family of sets when convenient. Given* $EV \subseteq E$ *we use* $EV_w \subseteq E_w$ *with* $w \in WIRE$ *or* $EV_c \subseteq E_c$ *with* $c \in COMP$ *to refer to the members of those families.*

Associated with every one-way interaction a there is one and only one event, $a\text{🔔}$. Each two-way interaction a has associated with it the set of five events $\{a\text{🔔}, a\text{✉}, a\checkmark, a\text{✗}, a\text{✞}\}$. Each event has a direction associated with it; an event is sent from one party to a co-party that receives it. For every two-way interaction a between party c and party c', the events $a\text{🔔}, a\checkmark, a\text{✗}$ and $a\text{✞}$ are sent by party c and received by party c', while the event $a\text{✉}$ is sent by c' and received by c. As it also described in 2.2, the events associated with a two-way interaction have specific roles and are correlated to each other. This correlation will be formalized further ahead. Also associated with the reply of two-way interactions there is a pledge that is guaranteed to hold within the deadline.

Definition 3 (Pledges). *The set* PP *of pledges is* $\{a.pledge : a \in 2WAY\}$*.*

The reply of a two-way interaction can be either negative or positive. In the last case there is a deadline before which the party that initiated the interaction can commit or cancel. We capture this through the notion of reply interpretation.

Definition 4 (Reply interpretation). *A reply interpretation* RI *assigns to every interaction* $a \in 2WAY$

- *a parameter* $a.reply^{RI} \in boolean_{\mathcal{U}}$*, indicating if the reply is positive.*
- *a deadline* $a.useBy^{RI} \in time_{\mathcal{U}}$ *for committing or cancelling.*

3.3 Computation States and Steps

As mentioned in 2.3, every wire has a time delay that defines the maximum time that an event takes to be delivered.

Definition 5 (Wire interpretation). *A wire interpretation Ψ assigns to every $w \in WIRE$ an element $w.delay^{\Psi} \in time_{\mathcal{U}}$.*

We will adopt a discrete state based model in which for every state of the system there are several possible activities each leading to a different state.

Definition 6 (Computation state)
A computation state for S is a tuple $\langle PND, INV, ENB, TIME, PLG, RI \rangle$ where:

- *$PND \subseteq E$ is the set of events pending in that state, i.e. the events that are waiting to be delivered by the corresponding wire.*
- *$INV \subseteq E$ is the set of events invoked in that state, i.e the events that have been delivered and are waiting to be processed.*
- *$ENB \subseteq E$ is the set of events that are enabled in that state, i.e. the events that will be executed if they are processed.*
- *$TIME \in time_{\mathcal{U}}$ is the time at that state.*
- *$PLG \subseteq PP$ the set of pledges that hold in that state.*
- *RI is a reply interpretation.*

In any state of the system there is a set of events that are pending in the wires, i.e. events that have been published, but haven't yet been delivered by the wires to the corresponding parties; this is represented by the set PND. INV is the set of events that were delivered by the wires and stored locally by each party where they are waiting to be processed. In any given state there is a set ENB of events that each party is ready to execute. Associated with each state there is also a time instant $TIME$, the set of pledges that are true in that state PLG and a reply interpretation for two-way events. The way the system changes from one state to another is given by the notion of computation step.

Definition 7 (Computation step)
A computation step for S is a tuple $\langle SRC, TRG, DLV, PRC \rangle^S$ where:

- *SRC and TRG are computation states*
- *$DLV \subseteq PND^{SRC}$ is the set of events that are selected for delivery during that step.*
- *PRC is a partial function that selects for each party c such that $INV_c^{SRC} \neq \emptyset$ an element of this set, i.e. it's the function that selects the event that will be processed.*
- *There is a set of actually-delivered events $ADLV \subseteq DLV$ such that for every $c \in COMP$:*
    - *If $PRC(c)$ is defined then $INV_c^{TRG} = (INV_c^{SRC} \setminus \{PRC(c)\}) \cup ADLV_c$*
    - *If $PRC(c)$ is undefined then $INV_c^{TRG} = INV_c^{SRC} \cup ADLV_c$*

- $PND^{TRG} = (PND^{SRC} \setminus DLV) \uplus PUB$ *where* $PUB \subseteq E$*, i.e. the events that were selected for delivery will no longer be pending in the target state; the new events that become pending in the target state are those that are published during the step*

For each step $\langle SRC, TRG, DLV, PRC \rangle$ *we also define the following set:*

- $EXC = \{PRC(c) : PRC(c) \in ENB_c^{SRC}\}$ *are the events that are executed during that step; those that are selected for processing and are enabled in the source state.*

The set of events that are pending in wires is updated during each computation step by removing the events that the wire delivers during that step — DLV — and adding the events that each party publishes — PUB. At each step, parties may choose to process one of the events waiting to be processed; this is captured by the function PRC. The fact each party can only process one event at a time is justified by the assumption that the internal state of the parties is not necessarily distributed and therefore no concurrent changes can be made to their states. We assume that not all of the events that are delivered are actually delivered to the receiving party; each wire may not be reliable, i.e. it may loose some of these events. The subset of delivered events that are actually delivered is given by $ADLV$. The set of events that are waiting to be processed in each party is updated in each step by removing the event that is processed and adding the events that are actually delivered to that party. The events that are executed on a computation step — EXC — are those that are processed during that step and are enabled in the source state.

Figure 6 is a graphical representation of the event flow during a computation step from the point of view of parties A and B connected by a wire W. Events $e \in INV_A$ and $e' \in INV_B$ that are waiting to be processed in the source state are selected for processing during the step ($PRC(A) = e$ and $PRC(B) = e'$) and therefore removed from these sets in the target state. The subset of pending events that is selected for delivery during the step is shown in light grey; some of these events are delivered to party A and enter the set INV_A while the rest are delivered to party B and enter INV_B. The set of events that are published by each party during the step is given by PUB_A and PUB_B; these events become pending in the wire in the target state. The notion of reliability for wires is given by the following definition:

Definition 8 (Reliable wire)
A wire w *is said to be reliable for a computation step if* $DLV_w = ADLV_w$*. The following property will necessarily hold for that step:*

- $DLV_w = ADLV_w = INV_w^{TRG} \setminus INV_w^{SRC}$.

That is, a wire is said to be reliable for a computation step if no event is lost by the wire on that step; each event in a reliable wire is either actually delivered to the destination party or it remains pending in the wire.

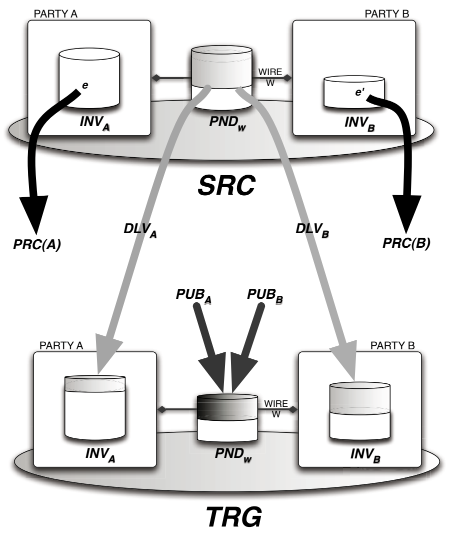

Fig. 6. A graphical representation of the event flow during a computation step from the point of view of a wire w between a pair of parties A and B. The system changes from state SRC to state TRG during the step.

3.4 Computation Trees

The different possible evolutions of a service-oriented system are given by a computation tree. In this paper we will consider only computations for which all wires are reliable for all steps.

Definition 9 (SRML tree)
A SRML tree for a signature S is of the form $\langle N, R, q_0, G\rangle$ where N is the set of nodes, $R \subseteq N \times N$ is the set of edges, $q_0 \in N$ is the root node and G is a labelling function that assigns a computation state to every node and a computation step to every edge. We use $n \longrightarrow n'$ to refer to an edge $(n, n') \in R$. Also, we use the following notation to refer to the elements of the labels:

- *If n is a node we use the names $PND^n, INV^n, ENB^n, TIME^n, PLG^n, RI^n$ to refer to the elements of the computation state $G(n)$ (in accordance with the names used in definition 6)*
- *If r is an edge we use the names $SRC^r, TRG^r, DLV^r, PRC^r, EXC^r, PUB^r$ to refer to the elements of the computation step $G(r)$ (in accordance with the names used in definition 7)*

Also, for every node $n \in N$ we define the set $UNPUB(n) = \{e \in E :$ there is no $r \in R$ such that $r < n$ and $e \in PUB^r\}$, i.e. the events unpublished between the root and node n. We use $<$ as a partial order relation on the sets of nodes and steps, $N \cup R$, based on the distance to the root node (e.g. $n < n'$ means that there is a path from the root node to n' that passes through n).

Not all trees represent valid evolutions. Many of the properties of service-oriented systems, described intuitively in section 2, like the sequence of events in a two-way interaction, concern the evolution of the system across several states. The definition of computation tree captures what are considered to be the valid models of service-oriented computation in SRML.

Definition 10 (Computation tree)
A computation tree for a signature S and a wire interpretation Ψ is a SRML tree $\langle N, R, q_0, G\rangle$ that satisfies the following rules:

Time elapsion. *For every edge $n \longrightarrow n'$, $TIME^n < TIME^{n'}$ (time moves forward)*

Single session. *For every event $e \in E$ if there is an edge $r \in R$ such that $e \in PUB^r$, i.e. e is published in r, then there is no edge $r' < r$ such that $e \in PUB^{r'}$ (events cannot be published more than once during a session — a computation tree models the evolution of a session).*

Wire delay. *For every event $e \in E_w$ and edge $r \in R$, if $e \in PUB^r$, i.e. if e is published in r, then for every subsequent node $n \in N$ such that $r < n$ and $TIME^{SRC^r} + w.delay^{\Psi} < TIME^n$, $e \notin PND^n$, i.e. e is not pending anymore in nodes where the time delay of the wire has elapsed (If an event is published then it will be delivered with a maximum delay)*

Event correlation. *For every two-way interaction $a \in 2WAY$, every node $n \in N$ and every edge $r \in R$ the following properties hold:*

1. *$a\boxtimes \in ENB^n$ if there is $r \in R$ such that $r < n$ and $a\clubsuit \in PUB^r$ and there is no $r' \in R$ such that $r' < n$ and $a\boxtimes \in EXC^r$ (the publication of the initiation-event enables the execution of the reply-event)*
2. *$a\clubsuit \in EXC^r$ iff for all $r < r'$ there is r'' such that either:*
   - *$r'' < r'$ and $a\boxtimes \in PUB^{r''}$ or*
   - *$r' < r''$ and $a\boxtimes \in PUB^{r''}$*

   (the reply-event of any interaction will be published after and only after the initiation-event was executed)
3. *If $a\boxtimes \in PUB^r$ then for every node $n, n', n'' \in N$ such that $r = n \longrightarrow n'$ and $n' < n''$, $RI^{n'} = RI^{n''}$ (The value of the reply, either positive or negative, and the associated deadline become fixed once the reply-event is published)*
4. *$a\checkmark$ and a✗ $\in ENB^n$ if:*
   - *there is an edge $r' \in R$ such that $r' < n$, $a\boxtimes \in PUB^{r'}$*
   - *$a.reply^{RI^n} = true$*
   - *there is no $r'' \in R$ such that $r'' < n$ and $a\checkmark \in EXC^{r''}$ or a✗ $\in EXC^{r''}$*
   - *$TIME^n < a.useBy^{RI^n}$*

   (the publication of a positive reply-event guarantees that the execution of the commit-event and the cancel-event becomes enabled until either one of them is executed or the deadline expires)
5. *$a.pledge \in PLG^n$ if the following conditions hold:*
   - *there is an edge $r' \in R$ such that $r' < n$ and $a\boxtimes \in PUB^{r'}$*
   - *$a.reply^{RI^n} = true$*

- *there is no* $r'' \in R$ *such that* $r'' < n$ *and* $a\checkmark \in EXC^{r''}$ *or* $a\text{✗} \in EXC^{r''}$
- $TIME^n < a.useBy^{RI^n}$

(The pledge must be true from the moment a positive reply is published until either the commit or the cancel are executed or the deadline expires)

6. *If* $a\checkmark \in PUB^r$ *where* $r = n \longrightarrow n'$ *then:*
   - *there is* $r' < r$ *such that* $a\boxtimes \in EXC^{r'}$
   - $a.reply^{RI^n} = true$
   - *there is no* $r'' < r$ *such that* $a\text{✗} \in PUB^{r''}$

   (The commit-event can only be published if the reply-event was executed, the reply was positive and the cancel-event was not published)
7. *If* $a\text{✗} \in PUB^r$ *where* $r = n \longrightarrow n'$ *then:*
   - $a.reply^{RI^n} = true$
   - *there is* $r' < r$ *such that* $a\boxtimes \in EXC^{r'}$
   - *there is no* $r'' < r$ *such that* $a\checkmark \in PUB^{r''}$

   (The cancel-event can only be published if the reply-event was executed, the reply was positive and the commit-event was not published)
8. *If* $a\text{✞} \in ENB^n$ *then there is* $r \in R$ *such that* $r < n$ *and* $a\checkmark \in EXC^r$ *and there is no* $r' \in R$ *such that* $r' < n$ *and* $a\text{✞} \in EXC^r$ *(the revoke-event can only be enabled after the execution of the commit-event)*
9. *If* $a\text{✞} \in PUB^r$ *then there is* $r' < r$ *such that* $a\checkmark \in PUB^{r'}$ *(The revoke-event can only be published after the commit-event was published)*

4 Concluding Remarks and Further Work

The primitives that we are proposing take into account proposals that have been made for Web-Service Conversation [5], in other modelling languages such as ORC [15], and in calculi such as Sagas [8]; they take into account that interactions are stateful and provide first-class notions such as reply, commit, compensation and pledge. The richness of the conversational model that we propose is reflected in the computational model. On the one hand, we need to account for the correlation that needs to be enforced among the different events involved in an interaction. On the other hand, we need to reflect the fact that events are transmitted through "wires" that enforce the interaction protocols that coordinate the joint behaviour of the parties involved in the delivery of the service.

The computational model we have defined captures the properties that are common to all SRML service-oriented systems independently of their specifications and any other interpretation constraints. SRML specifications have the role of defining the properties that are particular to a specific service-oriented system, i.e. restricting the set of trees that represent valid computations for that system. Further work is being carried out to give a complete formalization of the syntax and semantics of the SRML specification languages: the language of business roles, the language of business protocols and the language of interaction protocols. In this paper we have presented the work we have done so far towards formalising the semantic domain of these languages.

In connection with the previous we are working on applying the UCTL branching time temporal logic to the SRML framework. UCTL is an action/state based logic that was originally introduced to express the properties of UML statecharts [12]. The formal power that is attained with an action/state logic is crucial in order to reason about SRML models that, as we have seen in section 3, possess information related to the state of the system and the behaviour that changes the state. UCTL is also being used in other approaches to service-oriented computing: in [4] UCTL is used to reason about an asynchronous protocol for service-oriented applications; in [9] the UCTL framework is adapted in order to reason about a calculus for the orchestration of web services.

By defining the formal grounds over which UCTL can be applied to our models of service-oriented computation we accomplish several objectives. First we can validate the soundness of our computational model by defining the tools that allow us to reason about the model itself and axiomatize it. Second, these same tools will allow reasoning about service-oriented architectures both using proof strategies and automatic model-checking [12]. Finally, we lay the basis for defining the language of business protocols that is used to specify the behaviour of interfaces and that consists essentially of abbreviations of the UCTL temporal logic.

We have introduced time as a property of states, but gave no further insight into what kind of time model we will be using. We are currently investigating the best way to integrate time in our service-oriented model of computation taking into account the expressiveness, verifiability and model-checking requirements of the SRML framework [13].

Acknowledgements

Thanks are due to Antónia Lopes, Laura Bocchi, Stefania Gnesi and Franco Mazzanti for fruitful discussions.

References

1. Abreu, J., Bocchi, L., Fiadeiro, J.L., Lopes, A.: Specifying and Composing Interaction Protocols for Service-Oriented System Modelling. In: Derrick, J., Vain, J. (eds.) FORTE 2007. LNCS, vol. 4574, pp. 358–373. Springer, Heidelberg (2007)
2. Alonso, G., Casati, F., Kuno, H.A., Machiraju, V.: Web Services - Concepts, Architectures and Applications. Data-Centric Systems and Applications. Springer, Heidelberg (2004)
3. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
4. Beek, M., Fantechi, A., Gnesi, S., Mazzanti, F.: An action/state-based model-checking approach for the analysis of communication protocols for Service-Oriented Applications. In: Proceedings of the 12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2007), Berlin, Germany. LNCS, Springer, Heidelberg (2007)

5. Benatallah, B., Casati, F., Toumani, F.: Web service conversation modeling: A cornerstone for e-business automation. IEEE Internet Computing 8(1), 46–54 (2004)
6. Bocchi, L., Fiadeiro, J.L., Lopes, A.: The SENSORIA Reference Modelling Language: Primitives for configuration management (2006), `www.sensoria-ist.eu`
7. Bocchi, L., Hong, Y., Lopes, A., Fiadeiro, J.: From BPEL to SRML: A Formal Transformational Approach. In: Procedings of the 4th International Workshop on Web Services and Formal Methods (WSFM 2007). LNCS, Springer, Heidelberg (2007)
8. Bruni, R., Melgratti, H., Montanari, U.: Theoretical foundations for compensations in flow composition languages. SIGPLAN Not. 40(1), 209–220 (2005)
9. Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., Tiezzi, F.: A model checking approach for verifying COWS specifications. In: Proceedings of Fundamental Approaches to Software Engineering (FASE 2008). LNCS, Springer, Heidelberg (2007)
10. Fiadeiro, J.L., Lopes, A., Bocchi, L.: A Formal Approach to Service Component Architecture. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 193–213. Springer, Heidelberg (2006)
11. Fiadeiro, J.L., Lopes, A., Bocchi, L.: Algebraic Semantics of Service Component Modules. In: Fiadeiro, J.L., Schobbens, P.-Y. (eds.) WADT 2006. LNCS, vol. 4409, pp. 37–55. Springer, Heidelberg (2007)
12. Gnesi, S., Mazzanti, F.: A model checking verification environment for UML statecharts. In: Proceedings of XLIII Congresso Annuale AICA (2005)
13. Henzinger, T.A.: It's About Time: Real-Time Logics Reviewed. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 439–454. Springer, Heidelberg (1998)
14. Lazovik, A., Arbab, F.: Using Reo for Service Coordination. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 398–403. Springer, Heidelberg (2007)
15. Misra, J., Cook, W.: Computation orchestration: A basis for wide-area computing. Journal of Software and Systems Modeling (May 2006)
16. van der Aalst, W., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In: WS-FM, pp. 1–23 (2006)
17. Wirsing, M., Carizzoni, G., Gilmore, S., Gonczy, L., Koch, N., Mayer, P., Palasciano, C.: SENSORIA: A systematic approach to developing service-oriented systems — white paper (2007)