A Formal Model for Service-Oriented Interactions

José Fiadeiro[a], Antónia Lopes[b], João Abreu[c]

[a]*Department of Computer Science, University of Leicester, UK*
[b]*Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal*
[c]*Altitude Software, Portugal*

Abstract

In this paper, we provide a mathematical semantics for a fragment of a language — SRML— that we have defined in the IST-FET-GC2 Integrated Project SENSORIA for modelling service-oriented systems. The main goal of this research is to make available a foundational basis for the development of practical modelling languages and tools that designers can use to model complex services at a level of abstraction that captures business functionality independently of the languages in which services are implemented and the platforms in which they execute. The basic artefact of the language is the service module, which provides a model for a complex service in terms of a number of components that jointly orchestrate a business function and may dynamically discover and bind to external parties that can deliver required functionalities. We define a mathematical model of computation and an associated logic for service-oriented systems based on the typical business conversations that occur between the parties that deliver a service. We then define the semantics of SRML service modules over this model and logic, and formulate a property of correctness that guarantees that services programmed and assembled as specified in a module provide the business functionality advertised by that module. Finally, we define an algebraic operation of composition of service modules that preserves correctness. To the best of our knowledge, there is no other formal approach that has been defined from first principles with the aim of capturing the business nature of service conversations and support service assembly based on the business logic that is required, not as it is programmed.

Keywords: conversational protocols, formal methods, labelled transition systems, orchestration, service-oriented computing, temporal logic

1. Introduction

Service-oriented computing (SOC) is a paradigm for creating and providing business services via computer-based systems[1]. In SOC, services are computational entities that can be published together with a description of business functionality, discovered automatically and used by independent organisations to compose and provide new services (or for the internal use of that organisation). For example, a financial institution may provide a service for making secure electronic payments that can potentially be used by an insurance company to allow its clients to pay for car insurance or by an airline as part of its service for booking flights; a service for booking flights can in turn be used by a travel agency whose business is to arrange complete trips (i.e., transportation, accommodation, and so on).

Several technologies have been introduced by different stakeholders — e.g., OASIS (www.oasis-open.org) and W3C (www.w3.org) — with the goal of exploring this paradigm, for example in the form of Web services (e.g., [11, 61]) or Grid computing (e.g., [38]). A number of research initiatives have been proposing foundational theories, methods and formal techniques that address different aspects of SOC. Among them, the EU-funded research project SENSORIA [37] developed a number of well-founded analytical methods and tools for engineering service-oriented software. In this paper, we present the computational and coordination model, and associated logic, that support SRML [37] — the language that, in SENSORIA, was developed to support the design and analysis of services at a 'technology-agnostic' business level. We focus on the mechanisms that SRML offers, on the one hand, for *service assembly* — i.e., for specifying, at design time, how new services can be created by combining software components and interfaces to (external) services to be procured at run time — and, on the other hand, *service composition* — i.e., for composing a service specification that requires an external service with the specification of another service that matches the required interface. Although the term composition has been used fairly liberally in the literature, often in the sense of assembly as above, we will use it in this more restricted sense throughout the paper.

In the literature, two different approaches to service assembly have emerged: orchestration and choreography. In a nutshell, choreography (e.g., [62]) is

[1]The relationship between SOC and component-based software development or architecture description languages is discussed in [31, 33, 34]

concerned with the specification and realizability of a 'conversation' among a (fixed) number of peers that communicate with each other to deliver a service, whereas orchestration (e.g., [9]) is concerned with the definition of a (possibly distributed) business process (or workflow) that may use external services discovered and bound to the process at run time in order to deliver a service (accomplish a certain business goal). The two de facto standards for orchestration and choreography are WS-BPEL [2] (henceforth abbreviated as BPEL) and WS-CDL [58], respectively. BPEL is an OASIS standard that originated in a joint effort by IBM and Microsoft to define an XML-based language for programming business processes. BPEL addresses the orchestration of web services that are accessible through WSDL interfaces. WS-CDL is a standardisation candidate of W3C that can be used to specify a choreography of peers using WSDL (or XML schema) typed messages.

SRML is orchestration oriented, i.e., it addresses the notion of assembly that is supported by BPEL (a formal relationship between the two languages can be found in [17]). This also means that we adopt the two-party interaction model that is typical of the asynchronous, message-oriented middleware that supports SOC [9]: in SRML, all interactions involve only two parties and can be one-way (directed) or two-way (conversational), in which case they define a set of correlated messages exchanged by the two parties. Service-oriented systems can exhibit multi-party interactions but, in an orchestration model, these result from simpler, peer-to-peer connections. This is reflected in the mathematical model that we propose in Section 3, which offers a layer of abstraction in which these forms of interaction are 'native'.

Several calculi have been developed in the last few years that provide a mathematical semantics for the mechanisms that support choreography or orchestration — sessions, message/event correlation, compensation, inter alia (see [19] for a review on process calculi for SOC that provides an initial attempt to clarify the scope of existing proposals). Whereas such calculi address the need for specialised language primitives for *programming* in SOC, they are not abstract enough to understand the *engineering* foundations of SOC, i.e., those aspects (both technical and methodological) that concern the way applications can be developed to provide business solutions, independently of the languages in which services are programmed.

SRML was developed precisely to address that more abstract level of modelling — what we call 'business level', i.e., a level of modelling in which one can rely on the availability of the basic mechanisms mentioned above and address only those aspects that pertain to the application domain. Moreover,

our aim was not to compete with industrial standards but to put forward a simple modelling language with a clean mathematical semantics and associated logic that could serve as a proof-of-concept for formal modelling and analysis of service-oriented systems. Simplicity of the language is essential for a non-convoluted and workable mathematical semantics, but also for elegance. This is why we developed SRML from first principles rather than adopt or adapt an industry-strong language. The modelling language was validated over a number of case studies developed in the context of SENSORIA project and, for some of these case studies, realisations of the SRML models in BPEL and Java were developed in MSc projects. Details on the software engineering 'vision' that underlies our approach can be found in [37].

An overview of SRML is given in Section 2. In Section 3 we define a model of computation for service-oriented systems that is based on the typical business conversations that occur between the constituents of these systems. This model, which constitutes the semantic domain of SRML, is based on a notion of configuration of the global computers in which applications execute and get bound to other applications that offer required services. In Section 4 we give a logical characterisation of our model of computation using the logic UCTL [63, 64], i.e., we capture the properties that characterise computation in service-oriented systems and the requester and provider conversation protocols using temporal formulas. In Section 5 we provide a formal semantics for the specification languages used in SRML for the orchestration of services over our model of computation. In Section 6 we formalise the notion of service module, which in SRML is the mechanism for service assembly, and define an operation of composition of service modules. A notion of correctness is defined, which we show to be preserved by composition. In Section 7 we mention related work that can be found in the literature and discuss how it relates to the approach proposed herein.

Some aspects of this work, at earlier stages of development, have appeared in a number of papers (e.g., [5, 6]) and the PhD thesis [3], which also expands on a number of other aspects, notably the way model-checking techniques can be use for proving the correctness of service modules. This paper revises that earlier work to provide a formal model of service-oriented computation, interaction and orchestration. In particular, the notion of service composition and associated results as developed in Section 6 are presented here for the first time. A generic semantic model was developed in [36] for service discovery and binding, which can be instantiated to the particular computation model that we propose herein to produce a full model of ser-

vice orchestration, discovery and binding. An overview of SRML and the underlying approach to service-oriented modelling is given in [37].

2. Modelling services in SRML

The SENSORIA Reference Modelling Language (SRML) started to be developed within the SENSORIA project as a prototype domain-specific language for modelling services at a level of abstraction that is closer to business concerns. SRML is inspired by the Service Component Architecture (SCA) assembly model under 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"* [52]. While both SCA and SRML aim to support the middleware-independent layer that is concerned with the business logic of composite services, these two approaches have different practical goals:

- The main interest of SCA is to provide an open specification that allows the integration of several existing technologies (like JAVA, C++, WS-BPEL or PHP) for implementing the middleware-independent layer, i.e.,, the business logic;
- SRML focuses on providing a formal framework with a mathematical semantics for modelling and analysing the business logic of services independently of the hosting middleware, but also independently of the languages in which the business logic is programmed.

In this sense, SRML addresses a level of modelling that is more abstract than the one provided by the class of languages that have been developed for web services, e.g., WS-BPEL. The main novelty of SRML in addressing the business logic of services is that it adopts a set of primitives tailored specifically for modelling the business conversations that occur in SOC. In the world of SOC, service providers and their clients continuously engage in conversations — i.e., an exchange of correlated messages — with the goal of negotiating business deals. For example, a service requester may ask for a quote from a flight booking service on the basis of which it will decide either to go ahead with the deal and book the flight or cancel it. In SRML, services are characterised by the conversations that they support and the properties of those conversations. In particular, in SRML:

- the messages that are exchanged within a system are typed by their business function (requests, commitments, cancelations, and so on);
- the service interface behaviour is specified using message correlation patterns that are typical of business conversations; and
- the parties engaged in business applications need to follow pre-defined conversation protocols — requester and provider protocols.

The difference between SRML and more generic modelling languages is precisely in the fact that the mechanisms that support these conversation protocols, like message correlation, do not need to be modelled explicitly: they are assumed to be provided by the underlying SOA middleware. This is why SRML can be considered to be a domain-specific language: it frees the modeller from the need to specify aspects that should be left to lower levels of abstraction and concentrate instead on the business logic.

The conversation protocols supported by SRML also capture the role that time has in the outcome of the negotiations. For example, a flight agent may provide a quote for a given flight that is valid for two-minutes, after which seats may no longer be available or price may have changed.

Overall, SRML adopts a declarative style of modelling that promotes the development and maintenance of services based on business requirements, while abstracting from the way those services execute (see [65] for a discussion about the benefits of declarative specifications). The formal semantics of SRML, of which a part is defined in this paper, supports the design process with analytical techniques and tools [6, 13].

In this section we give a brief outline of the SRML language. We put the emphasis on the parts of the language that concern the business logic of service assembly and composition, which is the main focus of this paper.

2.1. Service assembly

The design of composite services in SRML adopts the SCA assembly model according to which new services can be created by interconnecting a set of elementary components to a set of external services [52]; the new service is provided through an interface to the resulting system. The business logic of such a service involves a number of interactions among these components and external services, but is independent of the internal configurations of the external services — the external services need only be described by their interfaces. The actual external services are discovered at run time by

matching these interfaces with those that are advertised by service providers (and optimising the satisfaction of service level agreement constraints).

The elementary unit for specifying service assembly and composition in SRML is the *service module* (or just *module* for short), which is the SRML equivalent to the SCA notion of "composite". A module specifies how a set of internal components and external required services interact to provide the behaviour of a new service. Figure C.5 shows the structure of the module $TravelBooking$, which models a service that manages the booking of a flight, a hotel and the associated payment. The service is assembled by connecting an internal component *BA* (that orchestrates the service) to three external services (for booking a flight, booking a hotel and processing the payment) and the persistent component *DB* (a database of users). The difference between the three kinds of entities — internal components, external services and persistent components — 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 [31]. By *TA* we denote the interface through which service requesters interact with *TravelBooking*. In SRML, interactions are peer-to-peer between pairs of entities connected through wires. *BP*, *BH*, *BF* and *BD* are the wires in $TravelBooking$.

2.2. Specifications

Each party (component or external service) is specified by declaring the interactions the party can be involved in and the properties that can be observed of these interactions during a session of the service. Wires are specified by the way they coordinate the interactions between the parties.

If the party is an internal component of the service (like *BA* in Figure C.5), its specification is an orchestration given in terms of state transitions — using the language of *business roles*, which we define in Section 5.1. An orchestration is defined independently of the language in which the component is programmed and the platform in which it is deployed; the actual component may be a BPEL process, a C++ or a Java program or a wrapped up legacy system, inter alia. An orchestration is also independent of the parties that are interconnected with the component at run time; this is because the orchestration does not define invocations of operations provided by specific

co-parties (components or external services); it simply defines the properties of the interactions in which the component can participate.

If the party is an external service, the specification is what we call a *requires-interface* and consists of a set of temporal properties that correlate the interactions in which the service can engage with its client. The language of *business protocols*, which we define in Section 5.2, is used for specifying the behaviour required of external services not in terms of their internal workflow but of the properties that characterise the interactions in which the service can engage with its client, i.e their interface behaviour. Figure C.6 shows the specification of the business protocol that the *HotelAgent* service is expected to follow. The specification of the interactions provided by the module (at its interface level) is what we call the *provides-interface*, which also uses the language of business protocols. Figure C.7 shows the specification of the business protocol that the composite service is expected to follow, i.e the service that is offered by the service module *TravelBooking*.

Persistent components can interact with the other parties synchronously, i.e., they can block while waiting for a reply. The properties of synchronous interactions are in the style of pre/post condition specification of methods. Because interactions of this kind have been well studied in the literature, e.g., in the context of component-based systems, we omit them from the paper and concentrate instead on the aspects that are intrinsic to SOC.

The specification of each wire consists of a set of *connectors* that are responsible for binding and coordinating, through *interaction protocols* as defined in Section 5.3, the interactions that are declared locally in the specifications of the two parties that the wire connects. Interactions are named differently in the two parties because composite services are put together at run time without a-priori knowledge of the parties that will be involved. Therefore, we need to rely on the interaction protocols of the wires to establish how these interactions are correlated.

The aspects that concern discovery and binding are addressed in SRML by a particular fragment of the language that is outside the scope of this paper. This fragment allows the specification of policies that define how the external services required by a module are selected and when they are bound. A formal semantics of such dynamic aspects is given in [36] in a way that is independent of the logic and languages that we use in this paper to specify the functional behaviour of services.

3. The semantic domain

In this section we present the semantic domain that underlies our modelling language. We model the execution of services using transition systems in which the transitions represent the exchange and processing of messages (events) by the parties involved in the delivery of the service.

3.1. Configurations of global computers

Our overall aim of SRML is to provide a modelling framework for service-oriented systems, by which we mean highly-distributed loosely-coupled applications that can bind to other applications discovered at run time and interact with them according to well-defined business protocols. In such a setting, there is no structure or 'architecture' that one can fix at design-time for a service-oriented system; rather, we can only rely on an underlying configuration of a global computer in which applications execute and get bound to other applications that offer required services. By global computers we mean "computational infrastructures available globally and able to provide uniform services with variable guarantees for communication, cooperation and mobility, resource usage, security policies and mechanisms" [1].

Given this, the semantic domain that we chose for SRML is not directed to individual applications. Instead, it addresses a more 'global' scenario in which (smaller) applications execute and respond to business needs by interacting with services and resources that are globally available. In this kind of domain, business processes can be viewed globally as emerging from a varying collection of loosely-coupled applications that can take advantage of the availability of services procured on the fly when they are needed.

Following on this motivation, the semantic domain that we put forward is based on the notion of *global configuration*, which we define as a simple graph whose nodes represent the various parties that are involved in given business processes and the edges (wires) represent the interconnections that exist between those parties and allow them to interact with each other. This implies that any given interaction involves only two parties, which is how service-oriented systems typically operate: they can exhibit multi-party interactions, but these result from simpler, peer-to-peer connections. The fact that the graph is simple — undirected, without self-loops or multiple edges — also means that all interactions between two parties are supported by a single wire and that no party can interact with itself. The graph is undirected because typical service-oriented interactions are conversational, i.e.,

the wires need to be able to transmit messages both ways.

Definition 3.1 (Global configuration). *A* global configuration *consists of a simple graph* $\langle$PARTIES, WIRES$\rangle$ *where* PARTIES *is the set of nodes and* WIRES *is the set of edges (the wires that connect the nodes). Note that each edge w is an unordered pair $\{n, m\}$ of nodes.*

Throughout the rest of this section, we assume a fixed global configuration $CONF = \langle$PARTIES, WIRES$\rangle$.

3.2. Interactions and events

In SOC, we are interested in systems that, in the context of a global configuration as defined above, consist of parties that interact by exchanging messages whose types are meaningful from the point of view of a business domain and which follow a conversational protocol that captures a business negotiation. This is is captured by the notion of *service execution configuration* — an extension of global configurations with information on the interactions that can take place between the parties during the execution of a service-oriented system.

Our approach focuses on patterns of message exchange that are typical in SOC, not so much on the data that is actually exchanged between the parties involved. Therefore, we abstract from any modelling aspects that relate to data and consider a fixed data signature $\Sigma = \langle D, F\rangle$ where D is a set of data sorts (such as *int*, *currency*, and so on) and F is a $D^* \times D$-indexed family of sets of operations over the sorts. We also assume a fixed partial algebra $\mathcal{U}$ for interpreting Σ. As motivated further on, partiality arises from the fact that interactions may have parameters whose values become defined only when certain events occur. We use $\perp$ to represent the undefined value and work with strong equality, i.e., two terms need to be defined in order to be equal. We further assume that $time, boolean \in D$ are sorts that represent the usual concepts of time and truth values, and that the usual operations over time and truth values are available in F.

Definition 3.2 (Service execution configuration). *A* service execution configuration *for CONF consists of a tuple $\langle INT, REPLY, USEBY\rangle$ where:*

- INT *is a* PARTIES$\times$PARTIES*-indexed family of mutually disjoint sets (of interactions) partioned into two families 2WAY (of two-way interactions) and 1WAY (of one-way interactions) . We require that, for every $n, n' \in$*PARTIES*, if $\{n, n'\} \notin$*WIRES *then $INT_{\langle n,n'\rangle} = \emptyset$.*

- *REPLY assigns to every a∈2WAY, a boolean constant a.*reply*.*
- *USEBY assigns to every a∈2WAY, a time-valued constant a.*useBy*.*

Interactions are indexed by ordered pairs of parties, which means that they are directional: an interaction a belonging to $INT_{\langle n,n'\rangle}$ is said to be *initiated* by the party n. Messages are transmitted through wires and, hence, the definition requires that there are no interactions between parties that are not connected by a wire.

We distinguish between one-way and two-way interactions. Two-way interactions are conversational: they involve a durative asynchronous exchange of correlated events (messages) that capture a pattern of dialogue that is prevalent in business applications: a party sends a request to a co-party that replies either positively and pledges to ensure a given property, or negatively, in which case the interaction ends. Given a two-way interaction a, we denote the reply by $a.reply$. If the reply is positive, the requester can commit to the deal or cancel the request. If (and after) the requester commits, a revoke action may be available to the requester whose effects are to compensate for the consequences of the commit action. The commit action can be performed from the moment a positive reply is given by the co-party until a deadline that we denote by $a.useBy$, after which the pledge expires. For example, a flight agent could state a price for every requested flight that is guaranteed to hold if the customer commits before 24 hours.

We use a particular notation to distinguish between the different types of events that can occur during interactions. The following table shows the events associated with a two-way interaction a:

a♠	The initiation-event of a.
a✉	The reply-event of a.
a✓	The commit-event of a.
a✗	The cancel-event of a.
a↟	The revoke-event of a.

One-way interactions capture situations in which a party sends a single message and does not expect a reply from the co-party. Therefore, there is one and only one event – a♠– associated with every one-way interaction a. The possible patterns of two-way interaction are depicted in Figure 1 for the case in which the reply is positive. For instance, using the example in Figure C.7, the service offered through $TravelAgent$ accepts a request for booking

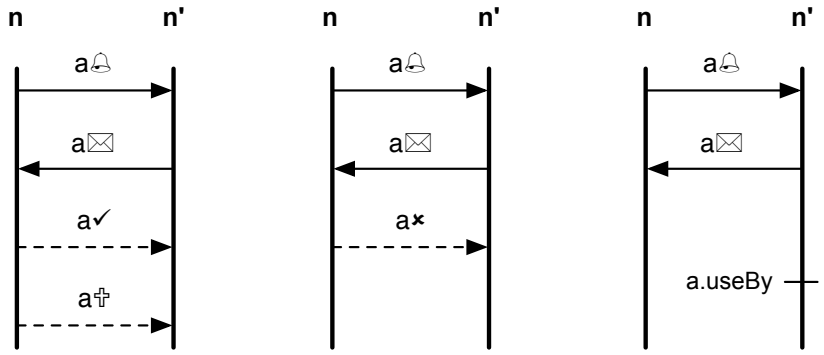


Figure 1: The patterns of two-way interactions that involve a positive reply. In the case on the left, the initiator commits to the deal; a revoke may occur later on, compensating the effects of the commit-event. In the middle, there is a cancellation; in this situation, a revoke is not available. In the case on the right, the deadline expires without a commit or cancel having occurred. Please note that the bell symbol appears in the text as 🔔.

a trip after a successful login has been communicated to the customer and the request arrives before the deadline offered to the customer; on receiving the reply to the request, the customer can either commit or cancel the deal; if the customer commits, the service notifies the customer when payment has been received, after which the customer can still revoke the deal until the day of departure (and receive a refund).

Because interactions are directional, the events associated with them have a sender and a receiver: for every $a \in 2WAY_{\langle n,n' \rangle}$, the events $a🔔$, $a✓$, $a✗$ and $a✝$ are sent by n and received by n', while the event $a✉$ is sent by n' and received by n. More precisely:

Definition 3.3 (Events). *For every $a \in INT$ and party $n \in$ Parties, the set $E_n(a)$ of* events *associated with a that can be received by n is as follows:*

- *If $a \in 2WAY_{\langle n,n' \rangle}$ then*
  - $E_n(a) = \{a✉\}$
  - $E_{n'}(a) = \{a🔔, a✓, a✗, a✝\}$
  - $E_{n''}(a) = \emptyset$ *for any other* $n'' \in$ Parties
- *If $a \in 1WAY_{\langle n,n' \rangle}$ then*
  - $E_n(a) = \emptyset$

– $E_{n'}(a) = \{a\text{🔔}\}$
– $E_{n''}(a) = \emptyset$ *for any other* $n'' \in$ PARTIES

We also define the following sets:

- $E_n = \bigcup_{a \in INT} E_n(a)$ *is the set of all the events that can be received by the party* n*.*
- *For every* $a \in INT_{\langle n,n' \rangle}$, $E(a) = E_n(a) \cup E_{n'}(a) = \{a\text{🔔}, a\text{✉}, a\text{✓}, a\text{✗}, a\text{⇧}\}$ *is the set of events associated with the interaction* a*.*
- $E = \bigcup_{a \in INT} E(a)$ *is the set of all events that can occur in the execution configuration.*
- *For every wire* $\{n, m\}$, $E_{\{n,m\}} = \bigcup_{a \in INT_{\langle n,n' \rangle} \cup INT_{\langle n',n \rangle}} E(a)$ *is the set of all the events that are carried by that wire.*

We see E *as a* WIRES*-indexed or a* PARTIES*-indexed family of sets when convenient.*

Throughout this section we consider a fixed service execution configuration $X_{CONF} = \langle INT, REPLY, USEBY \rangle$ for $CONF$.

3.3. Execution states, steps and models

In this subsection we define a model of computation for service execution configurations. In this model, each party publishes[2] or processes events. We take parties to be independent units of computation executing in parallel, which therefore can publish and process events simultaneously. As already stated, events are transmitted asynchronously by the wires. Once an event is delivered to the party to which it was sent, it is buffered until the party is ready to process it.

While it executes, every configuration generates a sequence of states, each of which is characterised by the events that are pending in wires, the events that are buffered by the parties, the current time, the history of event propagation, and an assignment of values to the parameters of the interactions (*useBy* and *reply*).

[2]We use the term "publish" to refer to the action of handing over an event to a wire — this is because in our model the routing of events is done entirely by the wires, i.e., the nodes are unaware of the destination of the events that they send.

Definition 3.4 (Execution state). *An* execution state *for X_{CONF} is a tuple $\langle PND, INV, TIME, HST, \Pi\rangle$ 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 wires.*
- *$INV \subseteq E$ is the set of events invoked in that state, i.e the events that have been delivered and are waiting to be processed.*
- *$TIME \in time_{\mathcal{U}}$ is the time at that state.*
- *HST is a family of four sets of events — $HST!$, $HST¡$, $HST?$ and $HST¿$ — which contain the events that have been published, delivered, executed and discarded, respectively.*
- *Π assigns to every interaction $a \in 2WAY$:*
  - *an instant of time $a.useBy^{\Pi} \in time_{\mathcal{U}}$;*
  - *a boolean $a.reply^{\Pi} \in bool_{\mathcal{U}}$.*

We use PND^s, INV^s, $TIME^s$, HST^s and Π^s to refer to the different components of an execution state s.

We introduce time in our model in order to be able to formalize the notion of deadline that is associated with two-way interactions and the fact that there is a transmission delay associated with wires. In [13], we go beyond this simple usage of time and present an approach to modelling and analysing real-time aspects of service-oriented systems based on our model, which addresses time-related aspects of quality of service.

State changes are performed during *execution steps*. More precisely, during each step, events can be *published* (i.e., handed over to the wires by the parties), *delivered* (i.e., handed over by the wires to the parties, which buffer them) or *processed* (i.e., taken from the buffers by the parties) in which case the life cycle of the event finishes. When processed, events can be either *executed* (which, typically, has effects on the state) or simply *discarded* (with no effects on the state in addition to the INV component of the state).

Definition 3.5 (Execution step). *An* execution step *for X_{CONF} is a tuple $\langle SRC, TRG, DLS, DLV, PRC, EXC, PUB\rangle$ where:*

- *SRC and TRG are the source and target execution states.*

- $DLS \subseteq PND^{SRC}$ *consists of events selected for delivery during the step.*
- $DLV \subseteq DLS$ *is the set of events effectively delivered during the step.*
- PRC *selects, for each party* n*, a subset of* INV_n^{SRC} *— the events that, having been invoked, are processed during the step (possibly none).*
- $EXC \subseteq PRC$ *is the set of events that are executed during the step; we denote by* DSC *the set of events that are discarded (processed but not executed), i.e.,* $DSC = PRC \setminus EXC$*.*
- $PUB \subseteq E$ *is the set of events that are published during the step. We require that* PUB *and* HST^{SRC} *be disjoint, i.e., the events chosen for publication must not have been published before.*
- $TIME^{SRC} < TIME^{TRG}$*, i.e., time moves forwards.*
- SRC *and* TRG *are such that:*
  - $INV_n^{TRG} = (INV_n^{SRC} \setminus PRC(n)) \cup DLV_n$ *i.e., the events that are processed during the step will no longer be waiting in the target state and the events that are* actually *delivered to a party will have to wait until they are processed.*
  - $PND^{TRG} = (PND^{SRC} \setminus DLS) \cup PUB$*, i.e., the events that are 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.*
  - $HST!^{TRG} = HST!^{SRC} \cup PUB$
  - $HST¡^{TRG} = HST¡^{SRC} \cup DLV$
  - $HST?^{TRG} = HST?^{SRC} \cup EXC$
  - $HST¿^{TRG} = HST¿^{SRC} \cup DSC$
  - *For every* $a \in INT$*,*
    - $\Pi(a.reply)^{TRG} = \begin{cases} \bot & \text{if } a✉ \notin HST!^{TRG} \\ \Pi(a.reply)^{SRC} & \text{if } a✉ \in HST!^{SRC} \end{cases}$
    - $\Pi(a.useBy)^{TRG} = \begin{cases} \bot & \text{if } a✉ \notin HST!^{TRG} \\ \Pi(a.useBy)^{SRC} & \text{if } a✉ \in HST!^{SRC} \end{cases}$

i.e., the values of a.reply and a.useBy remain undefined until set by the publication of the reply event, after which they remain unchanged.

That is to say, the set of events that are pending in wires is updated during a step by adding the events that each party publishes — PUB — and removing the events that the wire selects to deliver — DLS — during that step. The set of events that are waiting to be processed by each party is updated in each step by adding the events that are actually delivered — DLV — to that party and removing the events that have been processed — PRC. The history of events is updated at each step by adding to the corresponding subsets of HST the events that have been published, delivered, executed and discarded. Figure 2 illustrates the event flow during an execution step.

Three points in this definition need to be highlighted:

- The requirement that PUB and HST^{SRC} be disjoint (i.e., that events can not be published more than once) is justified by the fact that we model individual *sessions*. This stems from the fact that, as already discussed, our purpose is to provide support for the higher levels of abstraction of service modelling, which is based on a number of assumptions on the middleware over which services execute. Event correlation is one such mechanism that we rely on SOA middleware to provide, thus freeing the modeller from the need to handle event correlation across sessions explicitly (e.g., through parameters).

- The distinction between DLS — the events selected for delivery — and DLV — the events effectively delivered —allows us to model wires that are not *reliable*, i.e., not all events selected for delivery may actually reach the target parties. This is important if one wishes to provide analysis techniques over the reliability of interactions.

- The condition on time progression ($TIME^{SRC} < TIME^{TRG}$) does not mean that the step takes ($TIME^{TRG} - TIME^{SRC}$) to be executed. As a matter of fact, we consider that transitions are atomic and are recorded to occur at $TIME^{TRG}$.

The notion of *model* for an execution configuration can now be defined in terms of transition systems involving execution states and execution steps. Every such transition system captures the execution of the service during one

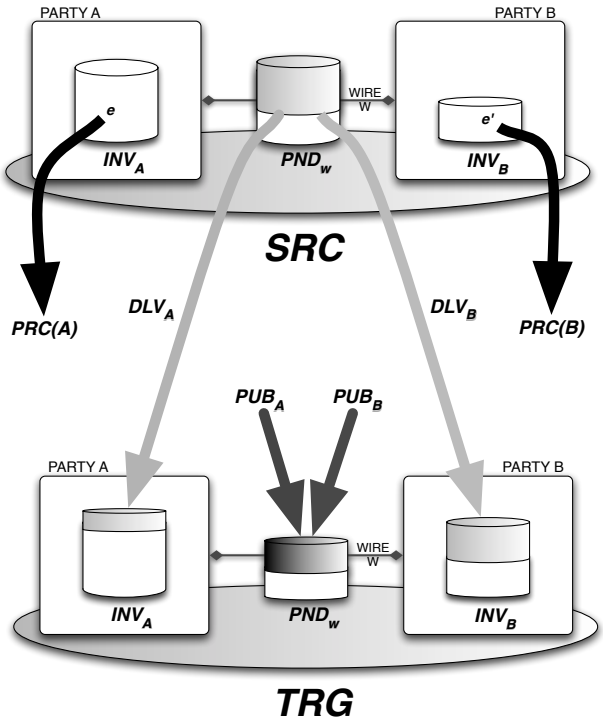


Figure 2: A graphical representation of the event flow during an execution step from the point of view of a reliable wire w between two parties A and B. The step makes a transition between states SRC and TRG. The set of events that are published by the two parties during the step is given by PUB_A and PUB_B; these events become pending in the wire 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 A and enter the set INV_A while others are delivered to B and enter INV_B. Events $e \in INV_A$ and $e' \in INV_B$ that are waiting to be processed in the source state are processed during the step ($e \in PRC(A)$ and $e' \in PRC(B)$) and therefore are not present in the target state.

session and defines the different choices that the service can make during that session. Each path in the transition system represents a possible execution in terms of the order and time at which events are published, delivered and processed. Notice that, because time is required to progress, these transition systems are actually directed acyclic graphs (DAGs).

We restrict ourselves to models that are *fair* in the sense that the events that are pending in the wires are eventually selected for delivery and that those that have been invoked and buffered are eventually processed. In order to define this notion of fairness more formally, it is convenient to agree on the following notational conventions.

Definition 3.6 (Path). *Let $\langle S, \rightarrow, s_0 \rangle$ be a transition system and $s \in S$. A* path *σ starting in s is:*

- *s — the* empty path *starting in s, or*
- *a possibly infinite sequence of transitions $(s \rightarrow s_1)(s_1 \rightarrow s_2) \ldots (s_{i-1} \rightarrow s_i) \ldots$ such that, if it is finite, its final state has no $\rightarrow$-successor states.*

We use $\sigma(i, i+1)$ to refer to the i-th transition in the sequence and $\sigma(i)$ to refer to the source state of this transition.

Definition 3.7 (Execution model). *A* model *Θ for X_{CONF} consists of a transition system $\langle S, \rightarrow, s_0 \rangle^{\Theta}$ where S is a set of execution states, $s_0 \in S$ is the initial state, and every transition $s \rightarrow s'$ is an execution step whose source and target are s and s', respectively. The transition system is required to be fair in the sense that, for every state $s \in S$ and event $e \in E$:*

- *If $e \in PND^s$ then, in every path starting in s, there is a transition r such that $e \in DLS^r$, i.e., every event that is pending in a wire is eventually selected for delivery;*
- *If $e \in INV^s$ then, in every path starting in s, there is a transition r such $e \in PRC^r$, i.e., every event that is invoked (i.e., buffered by a party) is eventually processed (executed or discarded).*

3.4. The conversation protocols

As discussed at the beginning of this section, a configuration defines which interactions can take place between which parties; in particular it defines which events each party can publish and which events it can receive. In the

case of a two-way interaction, one of the parties is defined in the configuration as the requester, meaning that it is able to publish the request, commit, cancel and commit-events and receive the reply, or as the provider, in which case it can publish the reply event and receive all the other events.

The notion of model given in Def. 3.7 captures the way a service executes during a session. We can now formalise the conversational protocol that we assume the SOA middleware to implement at each endpoint (requester and provider), as explained in Section 3.2 and illustrated in Figure 1.

In the following, we consider a fixed execution model Θ for X_{CONF}.

Definition 3.8 (Requester). *A party* $n \in$ PARTIES *is said to* behave as a requester *in an interaction* $a \in 2WAY_{\langle n,n' \rangle}$ *iff for every transition* $r = s \to s'$ *the following properties hold:*

1. *If* $a\checkmark \in PUB^r$ *then* $a\boxtimes \in HST?^{s'}$ *and* $a.reply^{\Theta^{s'}} = true$*, i.e., the commit-event can only be published with or after a positive reply.*
2. *If* $a\checkmark \in PUB^r$ *then* $a\text{✗} \notin HST!^{s'}$*, i.e., the commit-event can only be published if the cancel-event has not been published before (nor with the commit-event).*
3. *If* $a\text{✗} \in PUB^r$ *then* $a\boxtimes \in HST?^{s'}$ *and* $a.reply^{\Theta^{s'}} = true$*, i.e., the cancel-event can only be published with or after a positive reply.*
4. *If* $a\text{✗} \in PUB^r$ *then* $a\checkmark \notin HST!^{s'}$*, i.e., the cancel-event can only be published if the commit-event has not been published before.*
5. *If* $a\Lsh \in PUB^r$ *then* $a\checkmark \in HST!^{s}$*, i.e., the revoke-event can only be published after the commit-event has been published.*

To summarize, a requester can cancel or commit to the deal offered by the provider only after a deal has been offered — this is captured by clauses 1 and 3 — but it cannot both commit and cancel during the same session — this is captured by clauses 2 and 4. Finally, a requester cannot revoke a deal to which it did not previously commit — this is captured by clause 5.

The provider protocol is defined as follows:

Definition 3.9 (Provider). *A party* $n \in$ PARTIES *is said to* behave as a provider *in an interaction* $a \in 2WAY_{\langle n',n \rangle}$ *iff for every transition* $r = s \to s'$ *the following properties hold:*

1. *If* $a\spadesuit \in EXC^r$ *then either* $a\boxtimes \in PUB^r$ *or, in every path starting in* s'*, there is a transition* r' *such that* $a\boxtimes \in PUB^{r'}$*, i.e., a reply-event will be published when or after the initiation-event is executed.*

2. *If* $a\boxtimes \in PUB^r$ *then* $a\spadesuit \in HST?^{s'}$*, i.e., the reply-event can only be published after (or when) the initiation-event is executed.*
3. *If* $a\checkmark \in DLV^r$ *and* $TIME^{s'} < a.useBy^{\Theta^s}$ *then, in every path starting in* s'*, there is a transition* r' *such that* $a\checkmark \in EXC^{r'}$*, i.e., the commit-event will be executed if invoked before the deadline expires.*
4. *If* $a\boldsymbol{\times} \in DLV^r$ *and* $TIME^{s'} < a.useBy^{\Theta^s}$ *then, in every path starting in* s'*, there is a transition* r' *such that* $a\boldsymbol{\times} \in EXC^{r'}$*, i.e., the cancel-event will be executed if invoked before the deadline expires.*
5. *If* $a\Upsilon \in PRC^r$*, then* $a\checkmark \in HST?^{s'}$*, i.e., the revoke-event can only be processed after or with the execution of the commit-event.*

In summary, a provider always replies to a request, but it does so only if a request was made — this is captured by clauses 1 and 2 of the definition. A provider will be ready to execute a commit or a cancel (only one of the two) if invoked before the deadline expires — this is captured by clauses 3 and 4. Finally, a provider does not process a revoke-event before executing the associated commit — this is captured by clause 5.

Clause 5 is necessary to model long running transactions, in particular to guarantee that no request to revoke is unintentionally lost. The underlying assumption made by a provider is that if a revoke-event was received before the commit-event, it is probably because the wire delivered the events in the wrong order and not because the co-party made an unreasonable request — therefore a provider buffers every request to revoke until it has processed the associated commit. It is also important to notice that a provider may not necessarily accept a revoke; this is why no guarantee is given that the revoke will be executed when processed.

Definition 3.10 (Conversation-compliant model). *An execution model* Θ *is* conversation compliant *for a party* $n \in$ PARTIES *iff, for every* $n' \in$ PARTIES*,* n *behaves as a provider in every interaction* $a \in 2WAY_{\langle n',n \rangle}$ *and as a requester in every interaction* $a \in 2WAY_{\langle n,n' \rangle}$*.*

4. A logic of service interactions

In this section, we discuss how the logic UCTL [63, 64] can be used to specify and reason about properties of parties engaged in service execution models as defined in Section 3.

Several logics have been introduced in the past few years [25, 46, 49, 57] that capture both action-based and state-based properties, thus making it easier to formulate properties that, in pure action-based or pure state-based logics, can be quite cumbersome to write down. This is especially useful when logics are to be used in conjunction with languages and notations that — like the UML — allow both action and state changes to be expressed. In such cases, the use of combined action and state operators has the additional advantage of often leading to model checkers (e.g., UMC) that operate over a reduced state space and smaller memory, and spend less time during verification. UCTL subsumes both the branching-time action-based logic ACTL [27] and the branching-time state-based logic CTL [26]. This logic has been developed by ISTI-CNR in Pisa (our partners in the SENSORIA project) and used in conjunction with the UMC model checker to support the analysis of qualitative properties of services as reported in [3, 6].

Throughout this section, we assume a fixed global configuration $CONF = \langle \text{PARTIES}, \text{WIRES} \rangle$, an execution configuration X_{CONF} for $CONF$, and an execution model $\Theta = \langle S, \rightarrow, s_0 \rangle$ for X_{CONF}.

4.1. Interaction signatures

An interaction signature defines a set of interaction names that are used in the logic to identify actions that can occur during service execution (in particular, those that a specific party can perform). Interaction names are typed. For instance, Figure 3 depicts the interaction signature of the party FA, which declares three interaction names — *lockFlight* of type $r\&s$; and *payAck* of type rcv; and *payRefund* of type snd — and a number of parameters (discussed below). The types $s\&r$ and $r\&s$ are used for identifying two-way interactions from the point of view of the requester and provider, respectively, and snd and rcv for one-way interactions from the point of the sender and the receiver, respectively (see [17] for the way they correspond to the types of operations available in WSDL). For example, *lockFlight* identifies a two-way interaction in which FA acts as a provider.

We use $TYPE$ to designate the set $\{s\&r, r\&s, snd, rcv\}$.

Definition 4.1 (Interaction Signature). *An* interaction signature *is a pair* $\langle NAME, PARAM \rangle$ *where:*

- *$NAME$ is a $TYPE$-indexed family of finite and mutually-disjoint sets of interaction names;*

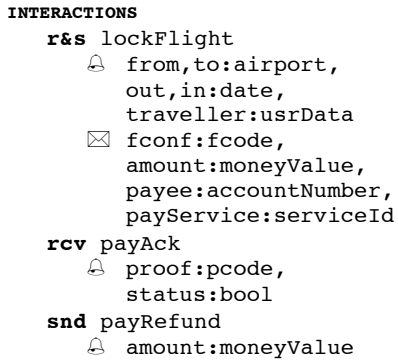


Figure 3: The interaction signature of party FA.

- *PARAM consists of five functions* $PARAM_{🔔}$, $PARAM_{✉}$, $PARAM_{✓}$, $PARAM_{✗}$ *and* $PARAM_{🗝}$ *such that:*
  - $PARAM_{🔔}$ *assigns to each name in NAME a D-indexed family of mutually disjoint sets of* 🔔*-parameters (associated with the initiation-event);*
  - $PARAM_{✉}$, $PARAM_{✓}$, $PARAM_{✗}$ *and* $PARAM_{🗝}$ *assign to each name* $a \in NAME_{s\&r} \cup NAME_{r\&s}$ *a D-indexed family of mutually disjoint sets of* ✉*-parameters,* ✓*-parameters,* ✗*-parameters and* 🗝*-parameters, respectively.*

Throughout the remainder of this section, we consider a fixed interaction signature $sig = \langle NAME, PARAM \rangle$.

As already mentioned, the types associated with the interaction names define the role the party plays, at execution time, in those interactions and, therefore, determine the actions that the party may execute. For example, BA, which is involved in the interaction *bookFlight* of type $r\&s$, can perform the action of publishing the reply-event of that interaction — denoted by *bookFlight*✉! — and the actions of receiving (i.e being delivered), executing and discarding the other events — denoted by *bookFlight*🔔¡, *bookFlight*🔔?, *bookFlight*🔔¿, and so on. Note that executing the reply-event of *bookFlight* is not an action of BA but of its co-party in the global configuration.

Definition 4.2 (Event names). *The NAME-indexed families* En^{PUB} *and* En^{RCV} *of sets of names of events that can be published and received, respectively, is defined as follows:*

If $a{\in}NAME_{s\&r}$ *then* $En_a^{PUB} = \{a\spadesuit, a\checkmark, a\text{✗}, a\Rsh\}$ *and* $En_a^{RCV} = \{a\text{✉}\}$

If $a{\in}NAME_{r\&s}$ *then* $En_a^{PUB} = \{a\text{✉}\}$ *and* $En_a^{RCV} = \{a\spadesuit, a\checkmark, a\text{✗}, a\Rsh\}$

If $a{\in}NAME_{snd}$ *then* $En_a^{PUB} = \{a\spadesuit\}$ *and* $En_a^{RCV} = \emptyset$

If $a{\in}NAME_{rcv}$ *then* $En_a^{PUB} = \emptyset$ *and* $En_a^{RCV} = \{a\spadesuit\}$

We define $En = En^{RCV} \cup EN^{PUB}$ *as the NAME-indexed family of sets of all event names associated with the signature sig.*

Definition 4.3 (Action Names). *The NAME-indexed families of sets of publication, delivery, execution and discard* action names *are defined as follows, where* $a{\in}NAME$*:*

$$Act_a^{PUB} = \{e! : e{\in}En_a^{PUB}\}$$

$$Act_a^{DLV} = \{e¡ : e{\in}En_a^{RCV}\}$$

$$Act_a^{EXC} = \{e? : e{\in}En_a^{RCV}\}$$

$$Act_a^{DSC} = \{e¿ : e{\in}En_a^{RCV}\}$$

We define $Act = Act^{PUB} \cup Act^{DLV} \cup Act^{EXC} \cup Act^{DSC}$ *as the NAME-indexed family of sets of all* action names *associated with the signature sig.*

Signatures are interpreted over service execution configurations as follows:

Definition 4.4 (Signature interpretation). *An* interpretation $\mathcal{I}$ *for sig over* X_{CONF} *is an injective function from NAME to INT such that:*

- *for every* $a{\in}NAME_{s\&r}{\cup}NAME_{r\&s}$, $\mathcal{I}(a){\in}2WAY$*, i.e., interaction names with type s&r or r&s denote two-way interactions;*
- *for every* $a{\in}NAME_{snd}{\cup}NAME_{rcv}$, $\mathcal{I}(a){\in}1WAY$*, i.e., interaction names with type snd or rcv denote one-way interactions;*

The interpretation extends to event names in the obvious way, i.e., for every $a{\in}NAME$ *and event of the form* $a\#$, $\#{\in}\{\spadesuit,\text{✉},\checkmark,\text{✗},\Rsh\}$, $\mathcal{I}(a\#) = \mathcal{I}(a)\#$*.*

It results that the types $s\&r$ and $r\&s$ are associated with the roles of requester and provider in two-way interactions, respectively. This is because the action names associated with $s\&r$ interactions denote the actions of a requester and the action names associated with $r\&s$ interactions denote the actions of a provider. Types *snd* and *rcv* are associated with the roles of sender and receiver in one-way interactions, respectively.

Signatures provide us with a way of specifying and reasoning about a particular view of the behaviour of a global configuration via the chosen interaction names. For specification purposes, we are particularly interested in signatures that are associated with specific parties, i.e., that define a complete view of the behaviour of a party. In order for an interaction signature to be associated with one party only, it is necessary that the interpretation names all and only the interactions involving that party.

Definition 4.5 (Local interaction interpretation). *An interaction interpretation $\mathcal{I}$ is said to be* local *to a party $n \in$ PARTIES iff:*

- *For every $a \in NAME_{s\&r} \cup NAME_{snd}$, $\mathcal{I}(a) \in INT_{\langle n,n' \rangle}$ for some party n', i.e., all interaction names with types s&r or snd denote interactions initiated by n;*
- *For every $a \in NAME_{r\&s} \cup NAME_{rcv}$, $\mathcal{I}(a) \in INT_{\langle n',n \rangle}$ for some party n', i.e., all interaction names with types r&s or rcv denote interactions initiated by some other party;*
- *For every $n' \in$ PARTIES, $INT_{\langle n,n' \rangle} \subseteq \mathcal{I}(NAME_{s\&r} \cup NAME_{snd})$ and $INT_{\langle n',n \rangle} \subseteq \mathcal{I}(NAME_{r\&s} \cup NAME_{rcv})$.*

4.2. Doubly-labelled transition systems

UCTL is interpreted over *doubly-labelled transition systems (L^2 TS)*, which are transition systems in which both the states and the transitions between states are labelled [28]. The L^2TSs used by UCTL differ from the classical notion of transition system in that they use sets of actions as labels rather than single actions. This allows systems to be modelled where several actions can take place simultaneously or a sequence of actions to be abstracted as a single state transition. For example, the reply-event of a two-way interaction may be published in the same transition during which the associated request-event is executed, as an effect of that execution.

Definition 4.6 (Doubly-labelled transition system). *A* doubly-labelled transition system *is a tuple* $\langle S, s_0, A, R, P, L\rangle$ *where:*

- S *is a set of states.*
- $s_0 \in S$ *is the initial state.*
- A *is a finite set of observable actions;*
- $R \subseteq S \times 2^A \times S$ *is the transition relation. We write* $s \xrightarrow{\alpha} s'$ *to denote a transition* $(s, \alpha, s') \in R$*;*
- P *is a set of atomic propositions;*
- $L : S \to 2^P$ *is a labelling function that associates with every state the set of all atomic propositions that are true in that state.*

In order to use UCTL for reasoning about services, we need to be able to extract a L^2TS from an execution model. The extracted L^2TS has the same structure as the model, i.e., they are defined over the same set of states and states are connected in the same way. In addition:

- The actions that label the L^2TS correspond to the different stages of event propagation as discussed in Section 3.
- Each state of the L^2TS contains information about the actions that have happened before the system reached that state, i.e., the history of event propagation. We need to make this information available directly because UCTL does not contain past operators (and the full expressive power of a logic with past operators such as [47] is not really necessary).

Definition 4.7 (L^2TS defined by a model). *The* L^2TS defined by an execution model Θ *is the tuple* $\langle S^\Theta, s_0^\Theta, A, R, P, L\rangle$ *where:*

- $A = Act$ *(cf. Definition 4.3);*
- *For every* $s, s' \in S$, $(s, \alpha, s') \in R$ *iff*
  - $s \rightarrow^\Theta s'$
  - $\alpha = \{e! : e \in PUB^{s \to s'}\} \cup \{e_! : e \in DLV^{s \to s'}\} \cup \{e? : e \in EXC^{s \to s'}\} \cup \{e_¿ : e \in DSC^{s \to s'}\}$

i.e., the transitions of the L^2TS are those of the execution model and are labelled with the publication (!), delivery (¡), execution (?) and discarding of events (¿);

- $P = \{e! : e{\in}E\} \cup \{e¡ : e{\in}E\} \cup \{e? : e{\in}E\} \cup \{e¿ : e{\in}E\}$;
- $L : S \to 2^P$ *is such that:*

  $$L(s) = \{e! : e{\in}HST!^s\} \cup \{e¡ : e{\in}HST¡^s\} \cup \{e? : e{\in}HST?^s\} \cup \{e¿ : e{\in}HST¿^s\}$$

  i.e., states are labelled by the history of event propagation.

From the definition of execution step (cf. Def. 3.5) it follows that actions cannot occur more than once:

Proposition 4.8 (Non-repetition). *Let $\langle S, \_, \_, R, \_, L\rangle$ be the L^2TS defined by an execution model, $s, s'{\in}S$ and $(s, \alpha, s'){\in}R$. For every $a{\in}\alpha$, $a{\notin}L(s)$.*

4.3. The language of service properties

The language of the logic that we use for specifying and reasoning about service properties extends UCTL with terms through which we can refer to the data that is transmitted between parties through interaction parameters. Throughout this section we consider a fixed signature *sig* and a fixed interaction interpretation $\mathcal{I}$ for *sig*.

Definition 4.9 (Terms). *The D-indexed family of sets TERM is defined inductively as follows:*

- *For every $d{\in}D$ and $c{\in}F_d$, $c{\in}TERM_d$.*
- *For every $d_1, ..., d_n, d_{n+1}{\in}D$, $f{\in}F_{<d_1,...,d_n,d_{n+1}>}$ and $\overrightarrow{p}{\in}TERM_{<d_1,...,d_n>}$, $f(\overrightarrow{p}){\in}TERM_{d_{n+1}}$.*
- *For every $d{\in}D$, $a{\in}NAME$ and $p{\in}PARAM(a)_d$, $a.p{\in}TERM_d$.*
- *For every $a{\in}NAME$, $a.useBy{\in}TERM_{time}$ and $a.reply{\in}TERM_{bool}$.*
- *$time{\in}TERM_{time}$.*
- *For every $act{\in}Act$, $act^t{\in}TERM_{time}$.*

By act^t we denote the time at which the action act was executed, i.e., when the corresponding event was published, delivered, executed, or discarded.

In order to be able to define the values that terms take over states, we need an interpretation function for interaction parameters:

Definition 4.10 (Parameter interpretation). *A* parameter interpretation *for a signature sig at a state $s \in S$ is an extension of the function Π^s that assigns a value $a.p^{\Pi^s} \in d_{\mathcal{U}}$ to every $d \in D$, $a \in NAME$ and $p \in PARAM(a)_d$. Furthermore, for every $a \in NAME$ and $p \in PARAM_{\#}$ for some $\# \in \{🔔, ✉, ✓, ✗, ⇧\}$ and transition $(s, \alpha, s') \in R$:*

$$a.p^{\Pi^{s'}} = \begin{cases} \bot & \text{if } a\# \notin L(s') \\ a.p^{\Pi^s} & \text{if } a\# \in L(s) \end{cases}$$

i.e., the values of the parameters remain undefined until set by the publication of the corresponding events, after which they remain unchanged.

We consider a fixed parameter interpretation that extends Π as above.

Definition 4.11 (Interpretation of terms). *The interpretation $[\![t]\!]_s$ of a term $t \in TERM$ in a state $s \in S$ of the L^2TS is defined as follows:*

- $[\![c]\!]_s = c_{\mathcal{U}}$
- $[\![f(t_1, ..., t_n)]\!]_s = f_{\mathcal{U}}([\![t_1]\!]_s, ..., [\![t_n]\!]_s)$
- $[\![a.p]\!]_s = a.p^{\Pi^s}$
- $[\![a.useBy]\!]_s = \mathcal{I}(a).useBy^{\Pi^s}$
- $[\![a.reply]\!]_s = \mathcal{I}(a).reply^{\Pi^s}$
- $[\![time]\!]_s = TIME^s$
- $[\![act^t]\!]_s = \begin{cases} \bot & \text{if } s = s_0 \\ TIME^s & \text{if } s' \xrightarrow{\alpha} s \text{ and } act \in \alpha \\ [\![act^t]\!]_{s'} & \text{if } s' \xrightarrow{\alpha} s \text{ and } act \notin \alpha \end{cases}$

Note that act^t is undefined in all states where act does not belong to their history.

Definition 4.12 (Action formulae). *The language of* action formulae *is defined as follows:*

$$\chi ::= true \mid t_1 = t_2 \mid act \mid \tau \mid \neg\chi \mid \chi \wedge \chi'$$

with $act \in Act$ *and* $t_1, t_2 \in TERM_d$ *for some* $d \in D$*. The satisfaction relation for action formulae is defined over the state transitions of the* L^2TS *as follows:*

- $s \xrightarrow{\alpha} s' \models true$
- $s \xrightarrow{\alpha} s' \models t_1 = t_2$ *iff* $[\![t_1]\!]_{s'} = [\![t_2]\!]_{s'}$*;*
- $s \xrightarrow{\alpha} s' \models act$ *iff* $\mathcal{I}(act) \in \alpha$*;*
- $s \xrightarrow{\alpha} s' \models \tau$ *iff* $\alpha = \emptyset$
- $s \xrightarrow{\alpha} s' \models \neg\chi$ *iff not* $s \xrightarrow{\alpha} s' \models \chi$
- $s \xrightarrow{\alpha} s' \models \chi \wedge \chi'$ *iff* $s \xrightarrow{\alpha} s' \models \chi$ *and* $s \xrightarrow{\alpha} s' \models \chi'$

Definition 4.13 (UCTL formulae). *The syntax of UCTL formulae is defined as follows:*

$$\begin{array}{rcl} \phi & ::= & true \mid act \mid t_1 = t_2 \mid \neg\phi \mid \phi \wedge \phi' \mid A\pi \mid E\pi \\ \pi & ::= & X_\chi \phi \mid \phi \,{}_\chi U \, \phi' \mid \phi \,{}_\chi U_{\chi'} \, \phi' \mid \phi_\chi W \phi' \mid \phi_\chi W_{\chi'} \phi' \end{array}$$

with $act \in Act$*,* $t_1, t_2 \in TERM_d$ *for some* $d \in D$ *and* χ, χ' *action formulae. We refer to* ϕ *as* state formulae *and to* π *as* path formulae*.*

UCTL formulae can be state formulae, meaning they are interpreted over states, or path formulae, meaning they are interpreted over paths. Note that action names are also used as state formulae, in which case they have a different meaning from their use as action formulae: a state satisfies an action name if the action that the name denotes has happened in the past.

A and E are *path quantifiers* and X, U and W are indexed *next*, *until* and *weak until* operators. The semantics of these operators is defined next. Their intuitive semantics is summarised in Appendix A.

Definition 4.14 (Satisfaction of UCTL formulae). *The satisfaction relation for UCTL formulae is defined as follows, where* s *is an arbitrary state and* σ *is an arbitrary path:*

- $s \models \mathit{true}$;
- $s \models \mathit{act}$ *iff* $\mathcal{I}(\mathit{act}) \in L(s)$;
- $s \models t_1 = t_2$ *iff* $[\![t_1]\!]_s = [\![t_2]\!]_s$;
- $s \models \neg\phi$ *iff not* $s \models \phi$;
- $s \models \phi \wedge \phi'$ *iff* $s \models \phi$ *and* $s \models \phi'$;
- $s \models A\pi$ *iff* $\sigma \models \pi$ *for all paths* σ *starting in* s;
- $s \models E\pi$ *iff there exists a path* σ *starting in* s *such that* $\sigma \models \pi$;
- $\sigma \models X_\chi \phi$ *iff* $\sigma(0,1) \models \chi$ *and* $\sigma(1) \models \phi$;
- $\sigma \models \phi_\chi U \phi'$ *iff there exists* $0 \leq j$ *such that* $\sigma(j) \models \phi'$ *and for all* $0 \leq i < j$, $\sigma(i) \models \phi$ *and* $\sigma(i, i+1) \models \chi$;
- $\sigma \models \phi_\chi U_{\chi'} \phi'$ *iff there exists* $1 \leq j$ *such that* $\sigma(j) \models \phi'$, $\sigma(j-1) \models \phi$ *and* $\sigma(j-1, j) \models \chi'$ *and for all* $0 < i < j$, $\sigma(i-1) \models \phi$ *and* $\sigma(i-1, i) \models \chi$.
- $\sigma \models \phi_\chi W \phi'$ *iff either*
  - $\sigma \models \phi_\chi U \phi'$; *or*
  - *for all* $0 \leq i$, $\sigma(i) \models \phi \wedge \neg\phi'$ *and* $\sigma(i, i+1) \models \chi$;
- $\sigma \models \phi_\chi W_{\chi'} \phi'$ *iff either*
  - $\sigma \models \phi_\chi U_{\chi'} \phi'$; *or*
  - *for all* $0 \leq i$, $\sigma(i) \models \phi$, $\sigma(i, i+1) \models \chi$, *and either* $\sigma(i) \models \neg\phi'$ *or* $\sigma(i, i+1) \models \neg\chi'$;

We also say that the pair $\langle \Theta, \mathcal{I} \rangle$ *satisfies a state formula* ϕ *iff* $s_0 \models \phi$, *where* s_0 *is the initial state of* Θ.

A number of derived operators that we use in the paper are collected in Appendix A. Examples of typical properties that are of interest for service-oriented systems are those that characterise the computational model itself and the requester and provider protocols, which can also be found in Appendix A.

5. The specification domains

In Section 3 we defined a semantic domain for the execution of (complex) services in terms of configurations of a number of parties that interact with each other by publishing and processing events according to a conversational protocol that is typical of business transactions. In this section, we define a language for modelling complex services over that semantic domain. More precisely, we provide a formal definition of the notion of *service module* that was introduced in Section 2 and of its semantics.

Throughout this section we consider:

- A global configuration $CONF = \langle \textsc{Parties}, \textsc{Wires} \rangle$;
- An execution configuration X_{CONF}=$\langle INT, REPLY, USEBY \rangle$ for $CONF$;
- An execution model Θ=$\langle S, \rightarrow, s_0 \rangle$ for X_{CONF};
- An interaction signature $sig = \langle NAME, PARAM \rangle$ where Act is the set of associated actions;
- An interaction interpretation $\mathcal{I}$ for sig extended to $PARAM$ as defined in 4.10.

As in previous sections, we consider a fixed data signature Σ=$\langle D, F \rangle$ and a fixed algebra $\mathcal{U}$ for Σ.

5.1. Business Roles: specifying components

Components are computational units that, together with the wires that connect them, execute a business workflow and orchestrate a (possibly empty) set of external services in order to offer a new service. Each component declared in a service module is typed by a (possibly underspecified) state machine — what we call a *business role*. A business role declares a set of state variables that model the state of the component and specifies a set of transitions that model the way state can change. A transition specification includes: (1) the trigger of the transition, which can be the execution of an event or an internal state change; (2) a guard, which defines a condition that must be satisfied for the trigger to cause the transition; (3) the effects that the transition has on the state of the component and the events that the component publishes during the transition. Transition specifications are

defined using a declarative textual language that permits underspecification, thus supporting an incremental design process.

An excerpt of the business role that specifies a *BookingAgent* is shown in Figure C.8. This business role contains the following state-variable declaration:

```
local s:[START, AUTHENTICATING, LOGGED, QUERIED, FLIGHT_OK,
      QUERIED, DATA_OK, FLIGHT_OK, HOTEL_OK, CONFIRMED,
      END_PAYED, END_UNBOOKED, COMPENSATING, END_COMPENSATED]
```

The state variable s ranges over a finite set of values (START, LOGGED, etc.) and is used in *BookingAgent* to model control flow. The other state variables are used for storing data that is needed at different stages of the orchestration. Formally:

Definition 5.1 (State variables). *A* state-variable declaration *VAR is a D-indexed family of disjoint sets. A* state-variable interpretation Δ *for VAR assigns to every state* $s \in S$ *and every state variable* $v \in VAR_d$ *an element* $v^{\Delta(s)} \in d_{\mathcal{U}}$ *(the value of the state variable on that state).*

Throughout this section we fix a state-variable declaration *VAR* and a state-variable interpretation Δ for *VAR*.

We now define the language of business roles over *sig* and *VAR*. First, we define the *sub-language of states* for specifying the state of the component. Then, we define the *sub-language of effects* for specifying the effects of state transitions. Finally, we define the notion of *transition specification* using the sub-languages of states and effects.

The sub-language of states is used for specifying the values of parameters, time and state variables in a state. It extends the notion of term given in Definition 4.9 with state variables:

Definition 5.2 (Language of States). *The D-indexed family STERM of sets of* state terms *is defined inductively through the rules used in Definition 4.9 and the additional formation rule: For every* $v \in VAR_d$, $v \in STERM_d$.

The interpretation $[\![t]\!]_s$ *of a state term* $t \in STERM$ *in* $s \in S$ *is defined as in Definition 4.11 extended with:* $[\![v]\!]_s = v^{\Delta(s)}$.

The sub-language of states *LS is defined as follows:*

$$\phi ::= true \mid t_1 = t_2 \mid \phi \wedge \phi' \mid \neg\phi$$

with $t_1, t_2 \in STERM_d$ *for some* $d \in D$. *The satisfaction relation is standard.*

In order to specify the effects of transitions, , namely how the state variables are updated and which events are published by the component, we need to extend this language with terms of the form v' where v is a state variable; as usual v' is used to denote the value that v has after the transition.

Definition 5.3 (Language of Effects). *The D-indexed family ETERM of sets of* effect terms *is defined inductively as in 5.2 with the following additional rules:*

- $time' \in ETERM_{time}$
- *For every* $v \in VAR_d$, $v' \in ETERM_d$

Effect terms are interpreted over transitions as in:

- $[\![a.p]\!]_{s_1 \to s_2} = \mathcal{I}(a).p^{\Pi^{s_2}}$
- $[\![v]\!]_{s_1 \to s_2} = v^{\Delta(s_1)}$, $[\![v']\!]_{s_1 \to s_2} = v^{\Delta(s_2)}$

The sub-language of effects LE is defined as follows:

- $\phi ::= true \mid t_1 = t_2 \mid pub \mid \phi \wedge \phi' \mid \neg\phi$

where $t_1, t_2 \in ETERM_d$ *for some* $d \in D$ *and* $pub \in En^{PUB}$. *The satisfaction relation is defined over transitions as in:*

- $r \models pub$ *iff* $\mathcal{I}(pub) \in PUB^r$

Definition 5.4 (Transition specification). *A* transition specification *is a triple* $\langle trigger, guard, effects \rangle$ *where* $trigger \in Act_a^{EXC} \cup LS$, $guard \in LS$, *and* $effects \in LE$. *The triple* $\langle \Theta, \mathcal{I}, \Delta \rangle$ *satisfies the transition specification iff, for every transition* $r = s_1 \to s_2$*:*

- *In the case where* $trigger \in Act_a^{EXC}$ *and* $\mathcal{I}(trigger) \in PRC^r$, *if* $s_1 \models guard$ *then* $\mathcal{I}(trigger) \in EXC^r$ *and* $r \models effects$, *otherwise* $\mathcal{I}(trigger) \in DSC^r$.
- *In the case where* $trigger \in LS$, *for every* $s_2 \to s_3$, *if not* $(s_1 \models trigger)$, $s_2 \models trigger$ *and* $s_2 \models guard$ *then* $s_2 \to s_3 \models effects$.

That is to say:

- In the case where the trigger is an event e, the event e is executed (i.e., not discarded) in every transition from a state in which the guard is true and during which e is processed, and its effects are observed in the target state.
- In the case where the trigger is not an event but a state condition, every transition that follows a transition that leads to a state in which the guard is true and during which the trigger condition becomes true (i.e., the condition is false in the source state, but true in the target state) satisfies the specified effects.

As an example, consider the following transition of the business role *BookingAgent*(see Figure C.8):

```
transition TripCommit
   triggeredBy bookTrip✓
   guardedBy s=HOTEL_OK
   effects s'=CONFIRMED
   sends bookFlight✓ ∧ bookHotel✓
      ∧ payment🔔
           ∧ payment.amount=bookFlight.amount
           ∧ payment.payee=bookFlight.payee
           ∧ payment.originator=getData.traveller
           ∧ payment.cardNo=getData.cardNo
```

This transition is triggered when the component processes the event *bookTrip*✓; if the component is in a state in which s has the value HOTEL_OK, then the transition is executed and, as a result, the events *bookFlight*✓, *bookHotel*✓ and *payment*🔔 are published, their parameters being set according to the stated constraints.

As discussed, a business role specifies a component by declaring the set of interactions in which that component can be involved, the set of state variables that characterise the internal state of the component and a set of transition specifications.

Definition 5.5 (Business role). *A* business role *is a triple* $\langle sig, VAR, ORCH \rangle$ *where sig is an interaction signature, VAR is a state-variable declaration and ORCH is a set of transition specifications for sig and VAR.*

The triple $\langle \Theta, \mathcal{I}, \Delta \rangle$ *satisfies the business role iff it satisfies every transition specification in ORCH.*

5.2. Business Protocols: specifying service interfaces

Service modules declare, through requires-interfaces, the types of external services that may need to be discovered and bound to the components that orchestrate the service provision. Each requires-interface is typed by a specification of the properties that the corresponding external service needs to satisfy regarding the way it interacts with the components declared in the module. Likewise, the provides-interface of the module is typed by a specification of the properties that the module offers at its interface. In SRML, these specifications are called *business protocols*.

A business protocol is an abstract description of a service that specifies the interactions in which the service can engage with its client and a number of properties of those interactions. Figure C.9 shows the business protocol of a *FlightAgent*. A business protocol is abstract in the sense that it does not specify the workflow of the service (as in business roles), but only a temporal correlation between the actions that the service performs. Throughout this section we consider $\langle S, s_0, Act', R, AP, L\rangle$ to be the L^{2}TS defined by the service execution model Θ (in the sense of Def. 4.7).

In principle, business protocols could be any set of state formulae. However, in order to facilitate the specification of service interface behaviour and the use of model-checking techniques, we followed [65, 66] and defined a set of *behaviour patterns* that capture requirements that are typical of service-oriented interactions — conditions under which events are executed or discarded (if processed) and conditions under which events are published. Each of the patterns is defined as an abbreviation of an UCTL formula. These patterns were validated over a number of case studies developed in the project SENSORIA, including telecommunications [4], financial [37], automotive [15] and procurement [35] scenarios.

In order to define these patterns, we make use of the subset of state formulae (cf. 4.13) that do not use temporal operators:

Definition 5.6 (State predicates). *The syntax of* state predicates *is defined as follows:*

$$\phi \;::=\; \mathit{true} \mid \mathit{act} \mid t_1 = t_2 \mid \neg\phi \mid \phi \wedge \phi'$$

with $\mathit{act} \in \mathit{Act}$ *and* $t_1, t_2 \in \mathit{TERM}_d$ *for some* $d \in D$.

The basic patterns are defined below. A set of derived patterns (used in the examples) and a diagrammatic explanation of their semantics can be found in Appendix B.

Definition 5.7 (Basic behaviour patterns). *Let* $e!, e?{\in}Act$, s, w *be state predicates and* a *an action formula. The* basic behaviour patterns *are:*

- "$s\ after\ a$" *stands for* $AG[a]s$ — *a always leads to a state in which s holds.*

- *"s enables $e?$ when g until w" stands for*

$$A(\neg s_{\neg e?} W s)$$
$$\wedge$$
$$AG(\ \ (\neg e? \wedge \neg s \wedge \neg w){:}$$
$$AX((s \wedge \neg w){:}A(((g{:}\langle e_{¿}\rangle false) \wedge \neg w)_{true} W w)))\ \ )$$
$$\wedge$$
$$AG(\ (\neg e? \wedge \neg s \wedge \neg w){:}AX(w{:}AG(\langle e?\rangle false))\ )$$
$$\wedge$$
$$\neg e? \ \ \wedge\ \ ((s \wedge \neg w){:}A(((g{:}\langle e_{¿}\rangle false) \wedge \neg w)_{true} W w))$$
$$\wedge$$
$$(w{:}AG(\langle e?\rangle false))$$

  — *e can only be executed during an interval defined by a state in which w is false and s becomes true and the first state before w becomes true. During this interval, e cannot be discarded if g is true. If there is no such interval, e will never be executed.*

- *"s ensures $e!$ before w" stands for*

$$A(\neg s_{\neg e!} W s)$$
$$\wedge$$
$$AG(\ \ (\neg e! \wedge \neg s \wedge \neg w){:}AX((s \wedge \neg w){:}A(\neg w_{true} U_{e!} true))\ \ )$$
$$\wedge$$
$$AG(\ (\neg e! \wedge \neg s \wedge \neg w){:}AX(w{:}AG(\langle e!\rangle false))\ )$$
$$\wedge$$
$$\neg e! \ \ \wedge\ \ ((s \wedge \neg w){:}A(\neg w_{true} U_{e!} true)) \ \ \wedge\ \ (w{:}AG(\langle e!\rangle false))$$

  — *after (and only after) s first becomes true, and provided w has been false, e will be published before w becomes true.*

Definition 5.8 (Business Protocol). *A* business protocol *consists of a pair* $\langle sig, BHV \rangle$ *where sig is an interaction signature and BHV is a set of behaviour patterns defined over sig.*

In order for a service execution model and signature interpretation to satisfy a business protocol, they must satisfy not only the specified patterns but also the set of UCTL formulae that characterise the behaviour of requesters or providers for each interaction declared to be of type $s\&r$ or $r\&s$, respectively, as discussed in Section 3.4. For example, a service of type *FlightAgent* (shown in Figure C.9) is requested to behave as a provider in the interaction *lockFlight*, i.e., it should always reply to a request, wait for a commit or cancel, and so on.

Definition 5.9 (Satisfaction of Business Protocols). *The pair* $\langle \Theta, \mathcal{I} \rangle$ *satisfies a business protocol* $\langle \langle NAME, PARAM \rangle, BHV \rangle$ *iff:*

- *For each* $a \in NAME_{s\&r}$, $\langle \Theta, \mathcal{I} \rangle$ *satisfies the requester protocol (cf. 3.8);*
- *For each* $a \in NAME_{r\&s}$, $\langle \Theta, \mathcal{I} \rangle$ *satisfies the provider protocol (cf. 3.9);*
- *For each* $b \in BHV$, $\langle \Theta, \mathcal{I} \rangle$ *satisfies the formula defined by b (cf. 5.7).*

5.3. Interaction Protocols: specifying wires

We have seen in the previous sections that the specification of the interactions a party is involved in is done locally for each party using local names and a local set of parameters for each interaction name. It is the responsibility of wire specifications to pair the interaction names that are used by two different wired parties to refer to the same (peer-to-peer) interaction and to correlate the parameters that are observed by each of the two parties for those interactions. In this section, we define a language for specifying wires, which includes the notion of *interaction protocol* — the specification primitive that we use for correlating pairs of interaction signatures.

Interaction protocols are similar to connector types in the sense of [8]. They consist of two roles (signatures) that provide abstract representations of the parties that can be connected, and a glue that specifies the protocol itself. Throughout this section we consider a fixed pair of interaction signatures sig_A and sig_B where $NAME_A$ and $NAME_B$ are disjoint.

Definition 5.10 (Interaction Protocol). *An* interaction protocol *consists of a triple* $\langle sig_A, sig_B, coord \rangle$ *where coord is a set of state formulae (cf. 4.13) for the union* $sig_A \cup sig_B$*, i.e., we consider terms and formulae that involve interactions from both signatures. We also consider a new kind of atomic state formulae that allow us to establish that two interaction names specify the same interaction:*

$$a = b$$

where $a \in NAME_A$ *and* $b \in NAME_B$*. We refer to* sig_A *and* sig_B *as the roles of the interaction protocol —* Role A *and* Role B*, respectively.*

We can also use this language to specify properties required of the wires. One such property is reliable delivery of published events. Another interesting class of properties concerns the bounds imposed on the delay with which wires transmit events. More complex properties may also be specified, for example data conversion between parameters, encryption mechanisms, and so on. In order to facilitate the specification of the two types of properties of wires mentioned above, we define the following two patterns:

Definition 5.11 (Wire patterns). *Let* $\langle sig_A, sig_B, coord \rangle$ *be an interaction protocol,* $a \in NAME_A \cup NAME_B$ *and* v *a term of sort time.*

- "*reliableOn a*" *stands for*

$$\bigwedge_{e \in En_a^{PUB}} AG[e!]AF_{e!}true$$

  — every event associated with a is eventually delivered after being published.

- "*a noLatterThan delay*" *stands for*

$$\bigwedge_{e \in En_a^{PUB}} AG[e!]AF_{e!}(time < e!^t + delay)$$

  — after being published, every event associated with a is delivered within the specified delay.

Figure C.10 shows an interaction protocol that connects an s&r interaction with a r&s interaction, where the initiation event (🔔) has three parameters and the reply event (✉) has just one. According to this protocol the

parameters of the two interactions are pairwise identical. Moreover, reliability in the delivery of events associated with the two interactions is required. The delay introduced by the wire in the transmission of these events is bound by the value v, which is a time value left undefined until the protocol is applied. Similarly, d_1, d_2, d_3 and d_4, which are the data sorts of the parameters, are also left undefined until the interaction protocol is applied. That is, $RStraight.I(d_1, d_2, d_3)O(d_4)D(v : time)$ stands for a family of interaction protocols.

Because we wish interaction protocols to be reusable, we define them independently of the names that are used for specifying the parties that they connect. In order to attach the roles of the interaction protocol to the interaction signatures of the two parties that we wish to connect we use *signature morphisms* (see Def. 5.12). A signature morphism μ from a signature s to a signature s' preserves the structure of interaction signatures: it maps each interaction name of s to an interaction name of s' with the same type, and each parameter of every interaction a to a parameter of $\mu(a)$ — choosing always a parameter that is associated with the same event, i.e., 🔔-parameters are mapped to 🔔-parameters, ✉-parameters are mapped to ✉-parameters, and so on. It is important to notice that not every interaction of the target signature needs to be in the image of a signature morphism. This allows for the coordination to address only part of the interactions as each party may interact with more than one co-party.

Definition 5.12 (Signature morphism). *A* signature morphism μ *from an interaction signature* $\langle NAME, PARAM\rangle$ *to another interaction signature* $\langle NAME', PARAM'\rangle$ *is a function that:*

- *assigns to each interaction name* $a{\in}NAME_t$ *and* $t{\in}TYPE$ *an interaction name* $a'{\in}NAME'_t$*;*
- *assigns to each parameter* $p{\in}PARAM_{\#}(a)_d$*, where* $\# \in \{🔔, ✉, ✓, ✗, 🗑\}$*, a parameter* $p'{\in}PARAM'_{\#}(\mu(a))_d$*.*

Interactions protocols are established between two parties using what we call *connectors*. A connector maps each of the roles of an interaction protocol to the signature of one of the two parties using a morphism (what is often called an 'attachment'). In that way, a connector defines which interactions are actually being coordinated by the interaction protocol. Figure C.11 shows the connector that binds the *BookingAgent* to the *HotelAgent* using the *RStraight* interaction protocol shown in Figure C.10.

Definition 5.13 (Connector). *A* connector *for two interaction signatures partyA and partyB is a triple* $\langle \mu_A, ip, \mu_B \rangle$ *where* $ip = \langle sig_A, sig_B, coord \rangle$ *is an interaction protocol, and* $\mu_A{:}sig_A \rightarrow partyA$ *and* $\mu_B{:}sig_B \rightarrow partyB$ *are injective signature morphisms.*

In order to define the semantics of connectors, we need to consider two interpretations $\mathcal{I}_A$ and $\mathcal{I}_B$ for *partyA* and *partyB*, respectively. In order to interpret the interaction protocol, we compose the morphisms with the interpretations, i.e., we use $(\mu_A; \mathcal{I}_A) \cup (\mu_B; \mathcal{I}_B)$ to interpret the interaction protocol. Notice that, because the signature morphisms are injective, the compositions with the interpretations are also injective and, therefore, so is the union, thus defining an interpretation (cf. 4.4).

Definition 5.14 (Satisfaction of Connectors). *Let* $\langle \mu_A, ip, \mu_B \rangle$ *be a connector for two interaction signatures partyA and partyB and* $\mathcal{I}_A$ *and* $\mathcal{I}_B$ *interpretations for partyA and partyB, respectively. The tuple* $\langle \Theta, \mathcal{I}_A, \mathcal{I}_B \rangle$ *satisfies the connector iff* $\langle \Theta, (\mu_A; \mathcal{I}_A) \cup (\mu_B; \mathcal{I}_B) \rangle$ *satisfies the set coord.*

Notice that atomic state formulae of the form $a = b$ where $a \in NAME_A$ and $b \in NAME_B$ are satisfied by $\langle \Theta, \mathcal{I}_A, \mathcal{I}_B \rangle$ iff

$$\mathcal{I}_A(\mu_A(a)) = \mathcal{I}_B(\mu_B(b))$$

i.e., the equation establishes that $\mu_A(a)$ of *partyA* and $\mu_B(b)$ of *partyB* name the same interaction.

Finally, we extend the notion of local interaction interpretation (cf. 4.5) to take into account the wires that connect two parties. Essentially, we want to be able to express that all interactions between two given parties are coordinated by a given connector:

Definition 5.15 (Local connector interpretation). *Let* $\langle \mu_A, ip, \mu_B \rangle$ *be a connector for two interaction signatures partyA and partyB. Let* $\mathcal{I}_A$ *and* $\mathcal{I}_B$ *be interpretations for partyA and partyB that are local to two parties* $p_A, p_B \in$ PARTIES*, respectively. We say that the pair* $\langle \mathcal{I}_A, \mathcal{I}_B \rangle$ *is local to* $\langle p_A, p_B \rangle$ *relative to* $\langle \mu_A, ip, \mu_B \rangle$ *iff* $\{p_A, p_B\} \in$ WIRES *and, for every* $a \in NAME_A$ *and* $b \in NAME_B$, $\mathcal{I}_A(\mu_A(a))$ *and* $\mathcal{I}_B(\mu_B(b))$ *belong to* $INT_{\langle p_A, p_B \rangle} \cup INT_{\langle p_B, p_A \rangle}$.

6. Service Modules

We can now give a formal definition of *service modules* as motivated in Section 2 and of their semantics. We also define a notion of composition of modules through which complex services can be defined from simpler ones.

6.1. Specifying services

A service module defines a local configuration labelled with specifications — business roles, business protocols, and connectors — and a *'provides-interface'* (a business protocol) for the clients that wish to use the service.

Definition 6.1 (Service module). *A* service module *L consists of:*

- *A local configuration $LOCAL{=}\langle \textsc{Parties}_L, \textsc{Wires}_L\rangle$ where:*
  - *$\langle \textsc{Parties}_L, \textsc{Wires}_L\rangle$ is a simple graph (as in 3.1)*
  - *The set of nodes is partitioned into two disjoint sets — a set $COMPS_L$ of* 'component-interfaces' *and a set $REQS_L$ of* 'requires-interfaces'*. The former correspond to the components that orchestrate the delivery of the service and the latter to the external services that may be required.*
- *$MAIN_L{\in}COMPS_L$ is a distinguished component responsible for the interactions with customers of the service.*
- *A labelling function $spec_L$ that assigns:*
  - *A business role to each $c{\in}COMPS_L$;*
  - *A business protocol to each $r{\in}REQS_L$;*
  - *A connector to each $w{=}\{p, p'\}{\in}\textsc{Wires}_L$ such that spec(w) is a connector for sign(p) and sign(p'), where by sign(n) we denote the signature of spec(n);*
- *A business protocol $provides_L{=}\langle sig, BHV\rangle$ (called the provides-interface) together with an inclusion morphism $sig{\rightarrow}sign(MAIN_L)$, i.e., sig is a sub-signature of $MAIN_L$.*

We further require that

- *for every $r{\in}REQS_L$ and $a{\in}NAME_{sign(r)}$, there is $n{\in}COMPS_L$ such that $\{n, r\}{\in}\textsc{Wires}_L$ and $a{=}\mu_i(b)$ for some $b{\in}NAME_{sig_i}$ (i=1,2), where $spec(\{n, r\}){=}\langle \mu_1, ip, \mu_2\rangle$ and $ip{=}\langle sig_1, sig_2, coord\rangle$.*
- *for every $c{\in}COMPS_L$ and $a{\in}NAME_{sign(c)}$, either $a{\in}NAME_{sig}$ or (exclusively) there is $n{\in}\textsc{Parties}_L$ such that $\{n, c\}{\in}\textsc{Wires}_L$ and $a{=}\mu_i(b)$ for some $b{\in}NAME_{sig_i}$ (i=1,2), where $spec(\{n, c\}){=}\langle \mu_1, ip, \mu_2\rangle$ and $ip = \langle sig_1, sig_2, coord\rangle$.*

The two conditions mean that, on the one hand, every interaction of every requires-interface needs to be connected, through a wire, to a component-interface of the module and, on the other hand, all interactions of component-interfaces except those of the provides-interface need to be connected to a party, i.e., all interactions must be internal to the module except those on the provides-interface. This is because we want all interactions with the environment to go through either the provides- or one of the requires-interfaces.

An example of a module is depicted in Figure C.5. Notice that, for simplicity, the interface *DB* to the persistent database is treated here as a requires-interface. In [37] we distinguish between requires-interfaces (for discoverable services) and uses-interfaces (for local resources). The distinction between the two is only relevant for the semantics of discovery and binding as explained in [36] — upon instantiation of a service module, new nodes and edges are added to the global-configuration graph for the component-interfaces and the wires that connect them, but not for uses-interfaces, which need to be identified among the parties already present in the global configuration (see below). The local configuration of this module contains only one component-interface — *BA*. Note that, although the figure suggests it, the provides-interface is not a party of the local configuration and its connection to *MAIN* is not a wire: they are depicted in that way in order to remain closer to the notation adopted in SCA [52].

Definition 6.2 (Interpretation structure). *An interpretation structure for a service module consists of:*

- *A global configuration* $CONF = \langle \textsc{Parties}, \textsc{Wires} \rangle$ *(cf. 3.1) and a homomorphism of graphs* $hom_L : LOCAL \rightarrow CONF$ *that is injective on both nodes and edges.*
- *An execution configuration* X_{CONF} *for CONF (cf. 3.2).*
- *An execution model* Θ *for* X_{CONF} *(cf. 3.7).*
- *For every party* $p \in \textsc{Parties}_L$*, an interaction interpretation* $\mathcal{I}_p$ *for the signature* $sign(p)$ *such that:*
  - *Every interaction interpretation* $\mathcal{I}_p$ *is local to p (cf. 4.4 and 4.5)*
  - *Every pair* $\langle \mathcal{I}_{p_A}, \mathcal{I}_{p_B} \rangle$ *is local to* $\langle p_A, p_B \rangle$ *relative to* $spec(\{p_A, p_B\})$ *(cf. 5.15)*

- *For every party* $p \in \textsc{Parties}_L$*, a parameter interpretation that extends the states of* Θ *as defined in 4.10.*
- *For every component* $c \in COMPS_L$*, a state-variable interpretation* Δ_c *(cf. 5.1).*

Notice that the homomorphism hom_L identifies the sub-configuration of the global configuration that corresponds to the execution of the service, which reflects the fact that we adopt an 'open' semantics, i.e., we consider the service as it executes in the context of a wider environment (global computer). For simplicity, we will take the homomorphism to be an inclusion whenever possible, in which case we refer just to $CONF$. As explained in [33, 36], this homomorphism plays an essential role in the process of dynamic reconfiguration that is induced by discovery and binding.

Definition 6.3 (Model of a service module). *An interpretation structure* $sem = \langle CONF, X_{CONF}, \Theta, \mathcal{I}, \Delta \rangle$ *is said to be a model for a module* m*, written* $sem \models m$*, iff:*

1. *For every party* $c \in COMPS_L$, $\langle \Theta, \mathcal{I}_c, \Delta \rangle$ *satisfies the business role* $spec(c)$ *(cf. 5.5);*
2. *For every party* $r \in REQS_L$, $\langle \Theta, \mathcal{I}_r \rangle$ *satisfies the business protocol* $spec(r)$ *(cf. 5.9);*
3. *For every* $w = \{p, p'\} \in \textsc{Wires}_L$, $\langle \Theta, \mathcal{I}_p, \mathcal{I}_{p'} \rangle$ *satisfies the connector* $spec(w)$ *(cf. 5.14);*
4. Θ *is conversation compliant for all parties (cf. 3.10)*

That is, a model is an interpretation structure that satisfies the specifications of all the parties and wires. Notice that, because parties and wires of $LOCAL$ are labelled with specifications, no properties are required of the behaviour of other parties (customers) that may interact with the service, except that they must behave as either a requester or a provider in all their interactions (which is captured by the fourth condition).

It remains to relate the provides-interface with the models of the module:

Definition 6.4 (Module correctness). *A service module* m *is said to be correct iff every model* $\langle \Theta, \mathcal{I} \rangle$ *of* m *satisfies the provides-interface of* m*, i.e.,* $\langle \Theta, \mathcal{I} \rangle \models provides(m)$.

Once again, notice that correctness establishes that any customer can rely on the properties offered through *provides*. This is because, in order for the module to be correct, all possible models (and, hence, all possible customers that are conversational compliant) need to be checked to satisfy those properties. In [6], we outlined an approach for establishing the correctness of service modules based on the use of the model-checker UMC [41]. UMC has been developed for the logic UCTL and UML statecharts. Our approach, which is fully detailed in [3], works for service modules whose business roles and interaction protocols are fully specified (in the sense that all underspecification has been removed), in which case they can be modelled as UML statecharts and, therefore, used by UMC. Naturally, the approach can only be applied to specifications that use the data types supported by UMC.

When the business roles and interaction protocols are coded using UML statecharts, the existence of a model satisfying 1–3 in Definition 6.3 is guaranteed. In that case, to establish the consistency of a module (i.e., the existence of a model) one only has to prove that these implementations are conversation compliant for all parties. For this purpose, we can resort to a number of analysis techniques that have been investigated and developed for service-oriented programming, most notably in the form of type systems for process calculi (e.g., [18, 51, 67]). Given the large amount of work that is now available for ensuring that interactions between distributed participants follow well-defined protocols, we decided to concentrate instead on higher-level properties, closest to the business-level, and rely on formal relationships with process calculi to perform that kind of analysis as discussed in [14] for the calculus COWS [51]. A similar mapping [13] was developed for the stochastic process algebra used by PEPA [40], which allows us to analyse timing properties of services.

Another interesting problem related to consistency is the existence of realisations of the specifications in a specific language (or multiple languages as in SCA [52]). In addition to UML statecharts, we have defined an encoding of WS-BPEL [17]. As discussed in Section 7, the advantage of the declarative language and the computational model that we chose for SRML is that they are simple and general enough to support realisations in different languages.

6.2. Composing modules

One of the key ideas of service-oriented computing is to modularise software applications by allowing them to procure external services and bind to them when they need them. The methodological and technical aspects

related to service discovery in our approach are presented in [36]. In this section, we show how our formal model supports an algebraic operation of module composition, i.e., the process of binding together a client service module with a provider one in order to create a new service module. Please notice that this is different from the process of creating a service module from business roles, business protocols and interaction protocols, which corresponds to service assembly.

Composition is based on matching the properties required by the client module with those offered by the provider module. Matching is based on the notion of logical consequence:

Definition 6.5 (Logical consequence). *Let $spec_r$ and $spec_p$ be sets of state formulae of UCTL over the same set A of observable actions and set P of atomic propositions. We say that $spec_r$ is entailed by (or is a logical consequence of) $spec_p$ iff, for every L^2TS $\langle S, s_0, A, R, P, L\rangle$, if $s_0 \models \phi$ for every $\phi \in spec_p$ then, for every $\phi \in spec_r$, $s_0 \models \phi$. That is, the properties of $spec_r$ are satisfied by every L^2TS that satisfies all the formulae of $spec_p$.*

Checking that a specification is entailed by another can be done using the proof techniques available for the logic concerned. In the specific case of business protocols, one of the motivations for the use of behaviour patterns was precisely to simplify this process of matching by standardising the specifications of required and provided behaviour, which should facilitate checking that required properties are matched by those provided. At present, the model-checking approach that we presented in [3, 6] can support matching by model-checking the patterns required by the client against the orchestration model of the provider. Research is still going onto to check pattern-based entailment based on theorem-proving techniques.

Consider then the situation in which we have two service modules

$$client = \langle LOCAL_c, MAIN_c, sem_c, provides_c\rangle$$

$$provider = \langle LOCAL_p, MAIN_p, sem_p, provides_p\rangle$$

where $provides_p = \langle sig_p, BHV_p\rangle$:

Definition 6.6 (Match). *A* match *for a requires-interface r_c of* client *with $spec_c(r_c)$=$\langle sig_c, BHV_c\rangle$ consists of an injective morphism η:$sig_c \rightarrow sig_p$ such that $\eta(BHV_c)$ is entailed by BHV_p.*

Notice that by $\eta(BHV_c)$ we denote the translation of UCTL formulas induced by the morphism η. This translation is obtained by applying the mappings on interaction names and parameters that define the morphism to the constituents of the formulae.

The following lemma will be used later on:

Lemma 6.7. *Consider a signature morphism μ:$sig_c \rightarrow sig_p$, an execution model Θ and a signature interpretation $\mathcal{I}$ for sig_p. Then, for every state formula ϕ in the language of sig_c, $\langle \mathcal{I}, s_0 \rangle \models \mu(\phi)$ iff $\langle \mu; \mathcal{I}, s_0 \rangle \models \phi$.*
Proof *The result is easily proved by induction on the structure of ϕ.* $\square$

Definition 6.8 (Composed module). *Given a match η for a requires-interface r_c of* client*, the composition of* provider *and* client*, which we denote by* (client $\oplus_{r_c,\eta}$ provider) *is defined as follows:*

- *Its configuration LOCAL is such that:*
  - $\textsc{Parties}_L = COMPS_L \cup REQS_L$ *is such that:*
    - $COMPS_L = COMPS_c \oplus COMPS_p$
    - $REQS_L = REQS_p \oplus REQS_c \backslash \{r_c\}$
  - $\textsc{Wires}_L = \textsc{Wires}_p$
    $\oplus \{\{n_c, MAIN_p\}: \{n_c, r_c\} \in \textsc{Wires}_c\}$
    $\oplus \{\{n_c, n'_c\}: \{n_c, n'_c\} \in \textsc{Wires}_c$ *and* $n_c, n'_c \neq r_c\}$
- $MAIN_L = MAIN_c$
- *Its labelling function $spec_L$ is such that:*
  - *For every* $n \in \textsc{Parties}_p$, $spec_L(n) = spec_p(n)$
  - *For every* $n \in \textsc{Parties}_c \backslash \{r_c\}$, $spec_L(n) = spec_c(n)$
  - *For every* $w \in \textsc{Wires}_p$, $spec_L(w) = spec_p(w)$
  - *For every* $w = \{n_c, n'_c\} \in \textsc{Wires}_c$ *s.t.* $n_c, n'_c \neq r_c$, $spec_L(w) = spec_c(w)$
  - *For every* $w = \{n_c, r_c\} \in \textsc{Wires}_c$, $spec_L(w) = \langle \mu_1, ip, \mu_2; \eta \rangle$ *where* $spec_c(w) = \langle \mu_1, ip, \mu_2 \rangle$
- *Its provides-interface $provides_L$ is $provides_c$.*

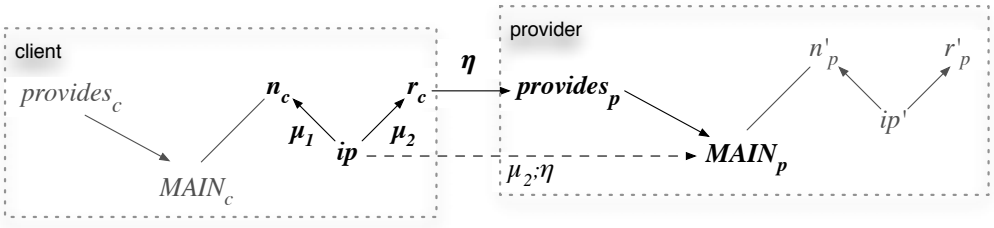


Figure 4: Composition of two modules

Proof *We need to check that the condition on the interactions of the requires-interfaces holds for the composition. This is straightforward because no new requires-interfaces are created and the connectors remain essentially the same (modulo the renaming through η).* □

That is to say, the configuration of the new module is obtained by amalgamating the configurations of *client* and *provider* identifying r_c with $MAIN_p$. The the main orchestration node of the new module is that of *client*. The nodes of the new module keep their labels (specifications) and so do the wires except for those that connect r_c in *client*. For those wires, we combine the attachment of the interaction protocol to r_c with η, i.e., by the signature morphism that defines the match, so that the attachment extends to $MAIN_p$. Notice that η actually maps to sig_p but sig_p is included in $sign(MAIN_p)$. This is depicted in Figure 4.

We will now prove a fundamental result: composition preserves correctness. In order to do so, we need to define a few auxiliary concepts and lemmas that allow us to generate models (reducts) for the client and the provider from a model of a composed module. We start with reducts for interpretations structures, first for the client:

Definition 6.9 (Client-reduct). *For every interpretation structure* $sem = \langle hom{:}LOCAL{\rightarrow}CONF, X_{CONF}, \Theta, \mathcal{I}, \Delta\rangle$ *of* (client $\oplus_{r_c,\eta}$ provider) *we define its client-reduct* sem_{client} *as follows:*

- $hom_c{:}LOCAL_c{\rightarrow}CONF$ *is such that:*
  - *For every* $n_c{\in}\textsc{Parties}_c\backslash\{r_c\}$, $hom_c(n_c) = hom(n_c)$
  - $hom_c(r_c) = hom(MAIN_p)$

  - *For every* w_c=$\{n_c, n'_c\}$∈WIRES$_c$ *s.t.* n_c,n'_c≠r_c, $hom_c(w_c) = hom(w_c)$
  - *For every* $\{n_c, r_c\}$∈WIRES$_c$, $hom_c(\{n_c, r_c\}) = hom(\{n_c, MAIN_p\})$
- *The service execution configuration* X_{CONF_c} *is* X_{CONF}.
- *The execution model* Θ_c *for* X_{CONF_c} *is* Θ.
- *The signature interpretation* $\mathcal{I}_c$ *is defined as follow:*
  - *For every* n∈PARTIES$_c$\$\{r_c\}$, $\mathcal{I}_{c_n} = \mathcal{I}_n$
  - $\mathcal{I}_{c_{r_c}}$=η;$\mathcal{I}_{MAIN_p}$
- *The parameter interpretation is defined in the same way.*
- *For every* n∈$COMPS_c$, Δ_{c_n}=Δ_c.

Lemma 6.10. *The client-reduct* sem_{client} *is an interpretation structure for the service module* client.

Proof *The proof is relatively straightforward. Essentially, we take the same elements as in the interpretation structure for the composition except in the case of elements involving the requires-interface,* r_c, *in which case we take corresponding elements that involve* $MAIN_p$. *Notice that, in the case of the interpretation of the interaction names and parameters of* r_c, *we need to map them to* $MAIN_p$ *through the signature morphism* η. *This corresponds to the intuition that the composition replaces the requires-interface of the client module with the provider module using* η *to translate the interaction names and parameters.*

The only point that is somewhat more subtle is the fact that $\mathcal{I}_{c_{r_c}}$ *is local to* r_c. *Consider then an interaction name* a∈$sign(r_c)$.

1. *From the definition of reduct,* $\mathcal{I}_{c_{r_c}}(a)$=$\mathcal{I}_{MAIN_p}(\eta(a))$.
2. *From the definition of service module, we know that there is* n∈PARTIES$_c$ *such that* $\{n, r_c\}$∈WIRES$_c$ *and* a=$\mu_i(b)$ *for some* b∈$NAME_{sig_i}$ *(i=1,2), where* $spec_c(\{n, r_c\})$=$\langle \mu_1, ip, \mu_2 \rangle$ *and* ip=$\langle sig_1, sig_2, coord \rangle$. *Let us take i=2 for the sake of argument.*
3. *From the definition of composition,* $\{n, MAIN_p\}$∈WIRES$_L$ *follows.*
4. *Because* $\langle \mathcal{I}_n, \mathcal{I}_{MAIN_p} \rangle$ *is local to* $\langle n$,$MAIN_p \rangle$ *relative to* $spec(\{n, MAIN_p\})$ *and* $spec(\{n, MAIN_p\})$=$\langle \mu_1, ip, \mu_2; \eta \rangle$, *we know that* $\mathcal{I}_{MAIN_p}(\eta(\mu_2(b)))$∈$INT_{\langle n, MAIN_p \rangle} \cup INT_{\langle MAIN_p, n \rangle}$.

5. *Hence,* $\mathcal{I}_{c_{r_c}}(a) \in INT_{c_{\langle n, rc\rangle}} \cup INT_{c_{\langle r_c, n\rangle}}$

The proof of locality for connectors follows the same line of reasoning. □

Now for the provider:

Definition 6.11 (Provider-reduct). *For every interpretation structure sem* $= \langle hom{:}LOCAL \rightarrow CONF, X_{CONF}, \Theta, \mathcal{I}, \Delta\rangle$ *of* (client $\oplus_{r_c,\eta}$ provider) *we define its provider-reduct* $sem_{provider}$ *as follows:*

- $hom_p{:}LOCAL_p \rightarrow CONF$ *coincides with hom on* $\textsc{Parties}_p$ *and* $\textsc{Wires}_p$.
- *The service execution configuration* X_{CONF_p} *is* X_{CONF}.
- *The execution model* Θ_p *is* Θ.
- *The signature interpretation* $\mathcal{I}_p$ *is the same as* $\mathcal{I}$ *restricted to the signatures of* provider*:*
- *The parameter interpretation is defined in the same way.*
- *The state-variable interpretation* Δ_p *is the same as* Δ *on* $COMPS_p$.

Lemma 6.12. *The provider-reduct* $sem_{provider}$ *is an interpretation structure for the service module* provider.
Proof *The result is immediate because sem and* $sem_{provider}$ *are essentially the same.* □

We now turn to models, starting with the provider:

Lemma 6.13. *Let sem be a model for* (client $\oplus_{r_c,\eta}$ provider). *The provider-reduct* $sem_{provider}$ *is a model for* provider.
Proof *The result is immediate because sem and* $sem_{provider}$ *are essentially the same and so are* $spec_L$ *and* $spec_p$. □

For the client-reduct to be a model, we need the provider to be correct:

Lemma 6.14. *Let sem be a model for* (client $\oplus_{r_c,\eta}$ provider)*. If* provider *is correct then* sem_{client} *is a model for* client*.*
Proof *The only property that is not immediately verified is the requirement that* sem_{client} *satisfies* BHV_c *where* $spec_c(r_c) = \langle sig_c, BHV_c \rangle$*. Given that* provider *is correct and, by lemma 6.13,* $sem_{provider}$ *is a model for* provider*,* $sem_{provider}$ *satisfies the provides interface* BHV_p *of* provider*. Because* η *is a match,* $sem_{provider}$ *satisfies* $\eta(BHV_c)$*. From lemma 6.7 and the fact that both* $sem_{provider}$ *and* $sem_{provider}$ *share the same execution model, we can conclude that* sem_{client} *satisfies* BHV_c*.* □

Finally, we can prove our main theorem:

Theorem 6.15 (Preservation of correctness). *Let* η *be a match for the requires-interface* r_c *of* client*. The composition* (client $\oplus_{r_c,\eta}$ provider) *is correct if* client *and* provider *are correct.*
Proof *Let* sem *be a model for* (client $\oplus_{r_c,\eta}$ provider)*. By lemma 6.14,* sem_{client} *is a model for* client*. Because* client *is correct,* sem_{client} *satisfies* $provides_c$*, which is the same as* $provides_L$*. Because sem and* sem_{client} *coincide in the way they interpret* $MAIN_L$*, which is the same as* $MAIN_c$*, it follows that sem satisfies* $provides_L$*.* □

7. Related work

Different formalisms and techniques have been brought to bear in order to meet the challenges raised by SOC. Among the approaches that address the modelling of service composition and assembly some focus directly on supporting the use of existing technological standards, while others (like SRML) remain largely 'technology agnostic'. The most popular formalisms that are being used for service assembly are those based on process algebras, Petri nets or automata and associated analysis techniques such as model-checking with temporal logic or bisimulation (in the case of process algebras). In this section we situate the work that is presented in this paper by discussing some of these formalisms.

7.1. Process algebras

Most of the applications of process algebras to SOC consist of extensions of well-established calculi (typically the π-calculus [55]) with primitives that

handle the mechanisms that are typical of SOA middleware. Because such calculi rely only on very few syntactic primitives, formal techniques have been defined that can be used very effectively for manipulating and reasoning about processes using techniques such as bisimulation. The disadvantage of having a very reduced syntax is that domain specific phenomena like the complex conversations that characterise SOC need to be encoded, making it harder to model applications at the level of the business logic.

For example, COWS (Calculus for Orchestration of Web Services) is a process algebra that provides operators inspired by BPEL [51]. In particular, COWS has a mechanism for correlating messages through the use of indexes. Verification techniques for COWS have been defined over an extension of UCTL and its associated model checker [32]. Model-checking COWS specifications requires an additional specification of the business semantics of the operations that are used in that COWS specification. This is because notions such as request, response, etc — which are native to SRML — are not part of COWS.

Boreale et al. propose SCC (Service Centered Calculus) [18], which is essentially an extension of the π-calculus with the same primitives for defining orchestrations that have been adopted by the ORC programming language for defining orchestrations [56]. ORC itself was originally a calculus for modelling orchestrations, but has evolved into a full-fledged programming language for implementing orchestrations [50]. SCC adopts a notion of service session for modelling client-server relationships. Session types [45] are also being brought forward to the realm of SOC (for example to track the types of the values exchanged in each session [7, 21]), and extended to capture the multi-party sessions that are typical of choreography [20, 24].

Vieira et al. [67] propose a variant of SCC that resorts to a generic notion of context for correlating messages that would otherwise be seen as independent. In SRML, two-way interactions define a much more complex notion of context, which is characterised by the events associated with them and the temporal correlation between those events.

Essentially, the difference between these formalisms and SRML is at the level of abstraction at which they operate. The aforementioned calculi provide a mathematical semantics for the mechanisms that support choreography or orchestration — sessions, message/event correlation, compensation, inter alia — but, just like any Java programmer does not need to program the dynamic allocation, referencing and de-referencing of names, a designer of a complex service should not need to be concerned with mechanisms that

are part of the *abstract operating system* that is offered by the SOA middleware. All that should be required from designers is that they identify and model the high-level business activities and the dependencies that they have on external services to fulfil their goals. Indeed, the main driver for the definition of SRML was not the lack of expressive power of existing calculi but the level of abstraction that they support. See [14] for a discussion on how these two levels of abstraction can be formally related.

7.2. Petri nets

Petri nets are a very well established formalism that lends itself to modelling a class of workflows — referred to as synchronisation workflows [48] — that contain the branching and synchronisation syntactic primitives found in BPEL [60], which has led to a number of semantics for BPEL (e.g., [60, 43, 54]). However, Petri nets have also been used in SOC independently of BPEL. Narayaman et al. [59] have implemented a tool for encoding OWL-S process descriptions into Petri nets so that analysis can be made using their simulation and modelling environment. The authors describe the composite behaviour of a service using a Petri net and discuss the conditions under which such behaviour can emerge by combining a set of web services. In our work, the problem is placed the other way around; we are interested in verifying if a fixed assembly of services behaves globally as intended.

Yi and Kochut propose an approach in which service interface conversations are modelled by WSDL-tailored Petri nets [68]. More precisely, each service interface is modelled by a Petri net where the inputs model WSDL operations and the connections between those inputs define a control flow that models the conversation. A set of such service interfaces can be composed by defining a Petri net that orchestrates them. The associated tools support the generation of BPEL code from composition specifications. It is also possible to generate from a composition specification a simplified WSDL tailored Petri net that models the interface conversation of the composite service — that Petri net can then be used in the assembly of other services.

Approaches that combine Petri nets with other formalisms also exist. For example, Hamadi et al. use Petri nets to model service behaviour and introduce a Petri net based algebra to model service assembly; the traditional algebraic techniques like bisimulation and structural equivalence can then be used to analyse service assemblies [42].

The main differences with respect to SRML are twofold. On the one hand, SRML offers a declarative style of specification that, in particular,

supports underspecification, i.e., modellers do not need to start from complete orchestrations but, instead, high-level designs that can be refined into more 'executable' specifications. On the other hand, and more importantly, SRML adopts an 'open systems' approach in that, in a module, the orchestration is not defined for a fixed collection of parties: the client is not fixed, and neither are the external services that may need to be discovered. The latter are represented by specifications of the behaviour that they are required to provide. This is reflected on the computational and coordination model that we have defined, and the way it supports the composition of modules by matching the requires-interface of a client with the provides-interface of a provider. In this sense, SRML goes beyond the scope of the usage of Petri-Nets for orchestrating a fixed collection of parties.

7.3. Automata and Temporal logic

Fu, Bultan et al. have investigated several aspects of service composition using guarded state machines with asynchronous communication to model the peers involved in a composite service. The authors analyse the problem of implementing a desired choreography, which they call a global conversation [23], with a system of such machines [22]. They also analyse the problem of verifying the correctness of an orchestration of a system of such guarded state machines [39] — in particular, the authors propose a tool-supported method for verifying the correctness of BPEL assemblies that involves translating BPEL processes into guarded state machines, translating those state machines into PROMELA (the input language of SPIN) and using the SPIN model-checker to verify the correctness of the composition (using Linear Temporal Logic) [44].

State machines were also adopted [11] to describe business protocols of services at an abstraction level similar to that we have in SRML. They extend state machines in order to include the specification of transition triggers (which might involve temporal and time constraints) and also the effects of transitions from the point of view of requesters (e.g., whether the transition is "compensatable" or is "definitive"). However, in contrast with SRML, data values exchanged during conversations are abstracted away. These business protocols are then used to generate skeletons of service implementations in BPEL [10]. Also based on state machines is the formalism proposed in [12] to specify web-service interfaces. In this case, it is possible to specify constraints over the exchanged data and to express temporal constraints on the ordering

of exchanged messages but time and features such as compensations and permanent effects are not covered.

Typically, temporal logic has been used within model-checking approaches to service-oriented system analysis for expressing the properties that a composite service is expected to satisfy, while the service itself is usually modelled using process algebras (e.g., [51]), Petri nets or automata (e.g., [63]). Some approaches focus on analysing real time properties of BPEL. For example, Diaz et al. model-check choreographies written with WS-CDL by translating them into timed-automata (i.e., timed state machines) — the authors advocate that the resulting (model-checked) automata can then be used for the automatic generation of correct BPEL orchestrations [29].

In SRML, we use temporal logic not only to express the properties that can be expected of a service (through the provides-interface of a module), but also to model the external services that are required to deliver those properties. This is essential for supporting dynamic discovery, selection and binding as detailed in [33, 36]. The applications of temporal logic mentioned above are essentially static in the sense that they are concerned with the way a fixed choreography can be realised by state machines. As defined in Section 6, SRML supports a notion of composition that is algebraic, i.e., we define an algebra of service modules in which temporal specifications are used for defining the conditions of composability.

As discussed in Section 4, SRML makes use of a set of patterns of temporal logic that capture commonly occurring requirements in business interactions. The use of patterns of logic for modelling system requirements has been addressed in the past (e.g., [53, 30]). Van der Aalst and Pesic have applied that idea and proposed DecSerFlow, a language for modelling services with patterns of Linear Temporal Logic (expressed graphically) that constrain the processes that the services follow, without the need to fully define these processes [65, 66]. The main difference between the patterns of logic used in SRML and those used in DecSerFlow derives from the fact that SRML is a domain-specific language; because of this, the abbreviations offered by SRML are tailored specifically for modelling business interactions, while those of DecSerFlow model generic processes.

8. Concluding remarks

A multitude of formal languages have been proposed for supporting the development of functionally correct services. A fundamental difference be-

tween those languages and our approach in general, and SRML in particular, derives from the fact that SRML is based on a set of constructs that capture what we believe are paradigmatic aspects of the business conversations that occur in SOC. More specifically, SRML offers, and our mathematical semantics addresses:

- messages that are typed by their business function;
- typical abstract message-correlation patterns;
- pre-defined conversation protocols.

Other approaches, some of which have been discussed in this paper, support the modelling of business conversations in a less specific way, adapting languages and formalisms that were developed for other purposes. For example, some calculi support conversations only in the sense that the set of messages can be partitioned into conversations (e.g., [51, 67]). Other approaches use generic workflow descriptions to model conversational protocols of services (e.g., [68]). As a result, modellers have to encode the specific mechanisms that SOC relies on — sessions, message/event correlation, compensation, inter alia — which should be left to the underlying SOA middleware to provide. To the best of our knowledge, there is no other formal approach that has been defined from first principles with the aim of capturing the business nature of service conversations and support service composition based on the business logic that is required not as it is programmed.

More precisely, SRML offers a higher level of abstraction in which complex services can be modelled independently of the specific languages that organisations such as OASIS (www.oasis-open.org) and W3C (www.w3.org) are making available for Web services (e.g., [11, 61]) or Grid computing (e.g., [38]). Our aim was to develop models and mechanisms that support the design of complex services from business requirements, and analysis techniques through which designers can verify or validate properties of composite services that can then be put together from (heterogeneous) service components using assembly and binding techniques such as the ones provided by SCA.

Towards this aim, SRML adopts a declarative style through which one can specify 'what' the components of a service do, while abstracting from 'how' they do it. Most other approaches focus on aspects of computation that are closely related to the way service composition is implemented, while SRML focuses on the logic of business integration and allows the executional

aspects to be left unspecified. This shift from procedural to declarative specifications for modelling services has been advocated, for example, by van der Aalst and Pesic [65] — who, nonetheless, have focused their efforts on monitoring the workflow of services at run time and have not addressed the issue of service design. For SRML, we have adopted the same principles but put forward a solution in which the business logic of composite services can be designed and analysed without forcing designers to make premature decisions about the way business logic is implemented.

The computational and coordination model that we proposed was used for supporting model-checking techniques that support qualitative analysis [6] and stochastic analysis techniques for timing properties of services [13]. Abstraction mappings from workflow languages (such as BPEL [17]), policy languages (such as StPowla [16]), and calculi (such as COWS [14]) have also been defined. Other essential aspects of SOC that SRML supports concern dynamic discovery, selection and binding of services, for which we developed an algebraic semantics that overlays the model proposed herein [33, 36].

Acknowledgements

We would like to thank our colleagues in the SENSORIA project for many useful discussions and feedback, especially Laura Bocchi, Stefania Gnesi and Franco Mazzanti who contributed directly to the work presented here. We would also like to thank the reviewers for having read the paper to such great depth and made so many helpful and challenging comments and suggestions.

References

[1] Global computing initiative. http://cordis.europa.eu/ist/fet/gc.htm.

[2] A. Alves et al. Web Services Business Process Execution Language Version 2.0, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[3] J. Abreu. Modelling Business Conversations in Service Component Architectures. PhD thesis. University of Leicester, 2009.

[4] J. Abreu, L. Bocchi, J. L. Fiadeiro, and A. Lopes. Specifying and Composing Interaction Protocols for Service-Oriented System Modelling. In *Proceedings of the 27th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, volume 4574 of *LNCS*, pages 358–373. Springer, 2007.

[5] J. Abreu and J. L. Fiadeiro. A Coordination Model for Service-Oriented Interactions. In *Proceedings of the 10th International Conference on Coordination Models and Languages (COORDINATION)*, volume 5052 of *LNCS*, pages 1–16. Springer, 2008.

[6] J. Abreu, F. Mazzanti, J. L. Fiadeiro, and S. Gnesi. A Model-Checking Approach for Service Component Architectures. In *Formal Techniques for Distributed Systems (FMOODS/FORTE)*, volume 5522 of *LNCS*, pages 219–224. Springer, 2009.

[7] L. Acciai and M. Boreale. A Type System for Client Progress in a Service-Oriented Calculus. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 642–658. Springer, 2008.

[8] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[9] G. Alonso, F. Casati, H. A. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, 2004.

[10] K. Baïna, B. Benatallah, F. Casati, and F. Toumani. Model-driven web service development. In *Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 3084 of *LNCS*, pages 290–306. Springer, 2004.

[11] B. Benatallah, F. Casati, and F. Toumani. Web services conversation modeling: A cornerstone for e-business automation. *IEEE Internet Computing*, 8(1):46–54, 2004.

[12] D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In *Proceedings of the 14th International Conference on World Wide Web (WWW)*, pages 148–159. ACM, 2005.

[13] L. Bocchi, J. L. Fiadeiro, S. Gilmore, J. Abreu, M. Solanki, and V. Vankayala. A formal approach to modelling time properties of service oriented systems. In *Handbook of Research on Non-Functional Properties for Service-Oriented Systems: Future Directions*, Advances in Knowledge Management Book Series, pages 36–60. IGI Global, in print.

[14] L. Bocchi, J. L. Fiadeiro, A. Lapadula, R. Pugliese, and F. Tiezzi. From architectural to behavioural specification of services. *Electr. Notes Theor. Comput. Sci.*, 253(1):3–21, 2009.

[15] L. Bocchi, J. L. Fiadeiro, and A. Lopes. Service-oriented modelling of automotive systems. In *Proceedings of the 32nd Annual IEEE International on Computer Software and Applications (COMPSAC)*, pages 1059–1064. IEEE, 2008.

[16] L. Bocchi, S. Gorton, and S. Reiff-Marganiec. Engineering service-oriented applications: From StPowla processes to SRML models. In *Proceedings of the 11th International Conference on Fundamental Aspects of Software Engineering (FASE)*, volume 4961 of *LNCS*, pages 163–178, 2008.

[17] L. Bocchi, Y. Hong, A. Lopes, and J. L. Fiadeiro. From BPEL to SRML: a formal transformational approach. In *Proceedings of the 6th International Workshop on Web Services and Formal Methods (WS-FM)*, volume 4937 of *LNCS*, pages 92–107. Springer, 2008.

[18] M. Boreale, R. Bruni, L. Caires, R. D. Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos, and G. Zavattaro. Scc: A service centered calculus. In *Web Services and Formal Methods*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006.

[19] R. Bruni. Calculi for service-oriented computing. In *Proceedings of the 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Web Services*, pages 1–41, 2009.

[20] R. Bruni, I. Lanese, H. Melgratti, and E. Tuosto. Multiparty Sessions in SOC. In *Proceedings of the 10th International Conference on Coordination Models and Languages (COORDINATION)*, volume 5052 of *LNCS*, pages 67–82. Springer, 2008.

[21] R. Bruni and L. G. Mezzina. Types and deadlock freedom in a calculus of services, sessions and pipelines. In *12th International Conference on Algebraic Methodology and Software Technology (AMAST)*, volume 5140 of *LNCS*, pages 100–115. Springer, 2008.

[22] T. Bultan and X. Fu. Specification of realizable service conversations using collaboration diagrams. *Service Oriented Computing and Applications*, 2(1):27–39, 2008.

[23] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *Proceedings of the 12th International Conference on World Wide Web (WWW)*, pages 403–410. ACM, 2003.

[24] M. Carbone, K. Honda, and N. Yoshida. A calculus of global interaction based on session types. *Electronic Notes in Theoretical Computer Science*, 171(3):127–151, 2007.

[25] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17(4):461–483, 2005.

[26] E. Clarke, E. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[27] R. De Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes*, pages 407–419, 1990.

[28] R. De Nicola and F. W. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.

[29] G. Diaz, J. Pardo, M. Cambronero, V. Valero, and F. Cuartero. Automatic Translation of WS-CDL Choreographies to Timed Automata. In *Formal Techniques for Computer Systems and Business Processes*, volume 3670 of *LNCS*, pages 230–242. Springer, 2005.

[30] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 411–420. ACM, 1999.

[31] A. Elfatatry. Dealing with change: components versus services. *Communications of the ACM*, 50(8):35–39, 2007.

[32] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying cows specifications. In *Proceedings of the 11th International Conference on Fundamental Aspects of Software Engineering (FASE)*, volume 4961 of *LNCS*, pages 230–245. Springer, 2008.

[33] J. L. Fiadeiro and A. Lopes. A model for dynamic reconfiguration in service-oriented architectures. In *Proceedings of 4th European Conference on Software Architecture (ECSA)*, pages 70–85, 2010.

[34] J. L. Fiadeiro and A. Lopes. An interface theory for service-oriented design. In *Proceedings of the 14th International Conference on Fundamental Aspects of Software Engineering (FASE)*, volume 6603 of *LNCS*, pages 18–33. Springer, 2011.

[35] J. L. Fiadeiro, A. Lopes, and L. Bocchi. A Formal Approach to Service Component Architecture. In *Proceedings of the 4th International Workshop on Web Services and Formal Methods (WS-FM)*, volume 4184 of *LNCS*, pages 193–213. Springer, 2006.

[36] J. L. Fiadeiro, A. Lopes, and L. Bocchi. An abstract model of service discovery and binding. *Formal Aspects of Computing*, 23(4):433–463, 2011.

[37] J. L. Fiadeiro, A. Lopes, L. Bocchi, and J. Abreu. The SENSORIA reference modelling language. In M. Wirsing and M. M. Hölzl, editors, *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *LNCS*, pages 61–114. Springer, 2011.

[38] I. Foster and C. Kesselman (Eds). The Grid 2: Blueprint for a new computing infrastructure, 2004.

[39] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proceedings of the 13th International Conference on World Wide Web (WWW)*, pages 621–630. ACM, 2004.

[40] S. Gilmore and J. Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In *Proceedings of the 7th International Conference Modelling Techniques and Tools for Computer Performance Evaluation*, volume 794 of *LNCS*, pages 353–368. Springer, 1994.

[41] S. Gnesi and F. Mazzanti. A model checking verification environment for UML statecharts. In *Proceedings of XLIII Congresso Annuale AICA*, 2005.

[42] R. Hamadi and B. Benatallah. A petri net-based model for web service composition. In *Proceedings of the 14th Australasian Database Conference - Volume 17*, volume 143 of *ACM International Conference Proceeding Series*, pages 191–200. Australian Computer Society, 2003.

[43] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to petri nets. In *Proceedings of the 3rd International Conference on Business Process Management (BPM)*, volume 3649 of *LNCS*, pages 220–235. Springer, 2005.

[44] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, May 1997.

[45] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *Proceedings of the 7th European Symposium on Programming Languages and Systems (ESOP)*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[46] M. Huth, J. Agadeesan, and D. Schmidt. Modal transition systems : A foundation for three-valued program analysis. In *Proceedings of the 10th European Symposium on Programming Languages and Systems (ESOP)*, volume 2028 of *LNCS*, pages 155–169. Springer-Verlag, 2001.

[47] M. Kaminski. A branching time logic with past operators. *J. Comput. Syst. Sci.*, 49(2):223–246, 1994.

[48] B. Kiepuszewski, A. H. M. ter Hofstede, and W. M. P. van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39(3):143–209, 2003.

[49] E. Kindler and T. Vesper. Estl: A temporal logic for events and states. In *Proceedings of the 19th International Conference on Application and Theory of Petri Nets*, volume 1420 of *LNCS*, pages 365–383. Springer-Verlag, 1998.

[50] D. Kitchin, A. Quark, W. R. Cook, and J. Misra. The Orc Programming Language. In *Formal Techniques for Distributed Systems (FMOODS/FORTE)*, volume 5522 of *LNCS*, pages 1–25. Springer, 2009.

[51] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *Proceedings of the 16th European Symposium on Programming Languages and Systems (ESOP)*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.

[52] M. Beisiegel et al. Service Component Architecture Specifications, 2007. http://www.osoa.org.

[53] Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. Technical Report CS-TR-90-1321, Stanford University, 1990.

[54] A. Martens. Analyzing web service based business processes. In *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 3442 of *LNCS*, pages 19–33. Springer, 2005.

[55] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[56] J. Misra. Computation orchestration: A basis for wide-area computing. In *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series II: Mathematics, Physics and Chemistry*, pages 285–330. Springer, 2005.

[57] M. Müller-Olm, D. A. Schmidt, and B. Steffen. Model-checking: A tutorial introduction. In *Proceedings of the 6th International Symposium on Static Analysis*, pages 330–354. Springer-Verlag, 1999.

[58] N. Kavantzas et al. Web Services Choreography Description Language Version 1.0, 2005. http://www.w3.org/2002/ws/chor/.

[59] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th International Conference on World Wide Web (WWW)*, pages 77–88. ACM, 2002.

[60] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Science of Computer Programming*, 67(2-3):162–198, 2007.

[61] C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, 2003.

[62] J. Su, T. Bultan, X. Fu, and X. Zhao. Towards a theory of web service choreographies. In *Proceedings of the 4th International Workshop on Web Services and Formal Methods (WS-FM)*, volume 4937 of *LNCS*, pages 1–16. Springer, 2007.

[63] M. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. 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)*, volume 4916 of *LNCS*. Springer, 2007.

[64] M. ter Beek, S. Gnesi, F. Mazzanti, and C. Moiso. Formal Modelling and Verification of an Asynchronous Extension of SOAP. In *Proceedings of the Fourth IEEE European Conference on Web Services (ECOWS)*, pages 287–296. IEEE Computer Society, 2006.

[65] W. van der Aalst and M. Pesic. DecSerFlow: Towards a truly declarative service flow language. In *Proceedings of the Third International Workshop Web Services and Formal Methods (WS-FM)*, volume 4184 of *LNCS*, pages 1–23. Springer, 2006.

[66] W. van der Aalst and M. Pesic. *Specifying and Monitoring Service Flows: Making Web Services Process-Aware*. Springer, 2007.

[67] H. T. Vieira, L. Caires, and J. C. Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In *Proceedings of the 17th European Symposium on Programming on Programming Languages and Systems (ESOP)*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.

[68] X. Yi and K. Kochut. A cp-nets-based design and verification framework for web services composition. In *Proceedings of the IEEE International Conference on Web Services (ICSW)*, pages 756–760. IEEE Computer Society, 2004.

Appendix A. Temporal operators

Intuitive semantics of the temporal operators

The operators A and E are interpreted over a state, while the remaining operators are interpreted over a path.

$A\pi$	All states of all paths from the current state satisfy π.
$E\pi$	There exists a path from the current state and state on that path that satisfies π.
$X_\chi\phi$	ϕ holds in the next state on the current path, which is reached by a transition that satisfies χ.
$\phi_\chi U\phi'$	Eventually, a state on the current path will be reached in which ϕ' holds and, until then, ϕ and χ will hold in every state and transition, respectively.
$\phi_\chi U_{\chi'}\phi'$	Eventually, a transition on the current path will be reached that satisfies χ' and leads to a state in which ϕ' holds; until then ϕ and χ will hold in every state and transition on the path, respectively.
$\phi_\chi W\phi'$	Either $\phi_\chi U\phi'$ holds in the current path or ϕ and χ will hold in every state and transition of the path, respectively.
$\phi_\chi W_{\chi'}\phi'$	Either $\phi_\chi U_{\chi'}\phi'$ holds in the current path or ϕ and χ will hold in every state and transition of the path, respectively.

Derived temporal operators

- $\langle\chi\rangle\phi$ stands for $E(X_\chi\phi)$
- $[\chi]\phi$ stands for $\neg\langle\chi\rangle\neg\phi$
- $EF\phi$ stands for $E(true_{true}U\phi)$
- $AF\phi$ stands for $A(true_{true}U\phi)$
- $AF_\chi\phi$ stands for $A(true_{true}U_\chi\phi)$
- $EG\phi$ stands for $E(\phi_{true}Wfalse)$
- $AG\phi$ stands for $A(\phi_{true}Wfalse)$

It is easy to conclude that:

$s \models AG\phi$ iff in every path σ from s and every $0 \leq i$, $\sigma(i) \models \phi$

$s \models [\chi]\phi$ iff in every path σ from s such that $|\sigma| \geq 1$, if $\sigma(0,1) \models \chi$ then $\sigma(1) \models \phi$

The intuitive semantics of the derived operators is captured in the following table:

$\langle\chi\rangle\phi$	There is a transition from the current state that satisfies χ and ends in a state in which ϕ holds.
$[\chi]\phi$	Every transition from the current state satisfies χ and ends in a state in which ϕ holds.
$EF\phi$	There is a path from the current state that leads to a state in which ϕ holds.
$AF\phi$	Every path from the current state leads to a state in which ϕ holds.
$EG\phi$	There is a path from the current state throughout which ϕ holds in every state.
$AG\phi$	ϕ holds in every state of every path from the current state

The axioms of service-oriented computation

In Section 3 we have presented our model of computation for service-oriented systems. We have defined which events can occur when a given configuration computes and how those events are transmitted between the nodes of the configuration. These properties can be expressed using formulae of UCTL, which are axioms in the sense that they hold in every execution model.

For every execution model Θ, signature interpretation $\mathcal{I}$, and $e \in En$, $\langle\Theta, \mathcal{I}\rangle$ satisfies:

Axioms of event propagation –

1. $A(true_{\neg e¡}W_{e!}true)$
   (Events can only be delivered after they are published)
2. $A(true_{(\neg e? \wedge \neg e¿)}W_{e¡}true)$
   (Events can only be processed after they are delivered)

3. $AG\neg(e? \wedge e¿)$
   (An event cannot be both executed and discarded)

Axiom of fairness –

1. $AG[e¡]AF_{(e? \vee e¿)}true$
   (After an event is delivered, it will eventually be processed, i.e., it will be executed or discarded)

Axioms of sessions –

1. $AG[e!]A(true_{\neg e!}Wfalse)$
   (An event can only be published once)
2. $AG[e¡]A(true_{\neg e¡}Wfalse)$
   (An event can only be delivered once)
3. $AG[e?]A(true_{\neg e?}Wfalse)$
   (An event can only be executed once)
4. $AG[e¿]A(true_{\neg e¿}Wfalse)$
   (An event can only be discarded once)

In addition to these axioms, which characterise the execution model, we can also define the axioms that characterise the conversational protocols, i.e., the behaviour of requesters and providers. The advantages of having a set of formulae that characterise these protocols is that it enables us to classify the nodes of a configuration by checking if they behave as requesters or providers for the two-way interactions in which they are involved.

For every execution model Θ and interpretation $\mathcal{I}$:

Axioms of requesters – A party $n \in$ PARTIES behaves as a requester in $a \in NAME_{s\&r}$, iff $\mathcal{I}(a) \in INT_{\langle n,n' \rangle}$ for some $n' \in$ PARTIES and $\langle \Theta, \mathcal{I} \rangle$ satisfies the following formulae:

1. $A(true_{\neg a\checkmark!}W_{(a\boxtimes ? \wedge a.reply)}true)$
   i.e., the commit-event will only be published after a positive reply was executed;
2. $AG[a\checkmark!]\neg a\text{✗}!$
   i.e., the commit-event will only be published if the cancel-event has not been published before;
3. $A(true_{\neg a\text{✗}!}W_{(a\boxtimes ? \wedge a.reply)}true)$
   i.e., the cancel-event will only be published after a positive reply was executed;

4. $AG[a\text{✗}!]\neg a\checkmark!$
   i.e., the cancel-event will only be published if the commit-event has not been published before;
5. $AG[a\text{↟}!]a\checkmark!$
   i.e., the revoke-event will only be published after the commit-event was published;
6. $AG[a\text{↟}! \wedge a\checkmark!]false$
   i.e., the revoke-event and the commit-event cannot be published simultaneously.

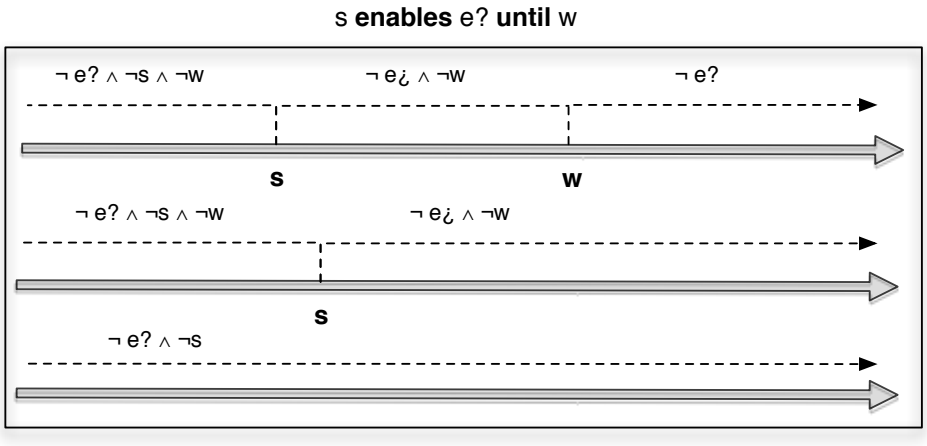
Axioms of providers – A party $n \in$ PARTIES behaves as a provider in $a \in NAME_{r\&s}$, iff $\mathcal{I}(a) \in INT_{\langle n',n\rangle}$ for some $n' \in$ PARTIES and $\langle \Theta, \mathcal{I}\rangle$ satisfies:

1. $AG[a\text{🔔}?]AF_{a\text{✉}!}true$
   i.e., after the initiation-event is executed a reply-event will be published;
2. $A(true_{\neg a\text{✉}!}Wa\text{🔔}?)$
   i.e., the reply-event will only be published after (or when) the initiation-event is executed;
3. $AG[a\checkmark\text{¡} \wedge time < a.useBy]AF_{a\checkmark!}true$
   i.e., a commit-event will be executed if delivered before the deadline expires;
4. $AG[a\text{✗}\text{¡} \wedge time < a.useBy]AF_{a\text{✗}!}true$
   i.e., a cancel-event will be executed if delivered before the deadline expires;
5. $AG[a\text{↟}? \vee a\text{↟¿}]a\checkmark?$
   i.e, the revoke-event will not be processed before executing the commit-event.

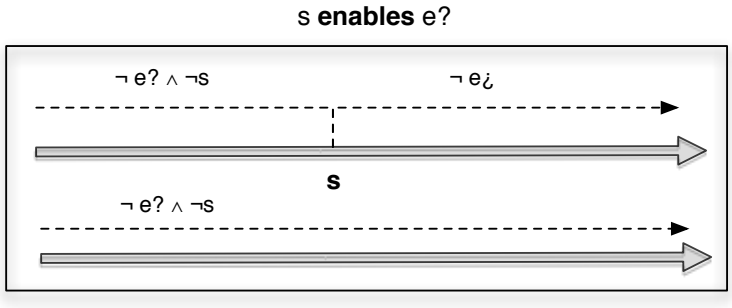
Appendix B. Derived behaviour patterns

The following patterns are particular instances of the basic ones defined in Section 5.2 and are used in the examples:

- "*s enables e? until w*" abbreviates "*s enables e? when true until w*" — *e* can only be executed, and cannot be discarded, after *s* (first) becomes true but only while *w* has never been true. After *w* (first) becomes true *e* cannot be executed anymore.

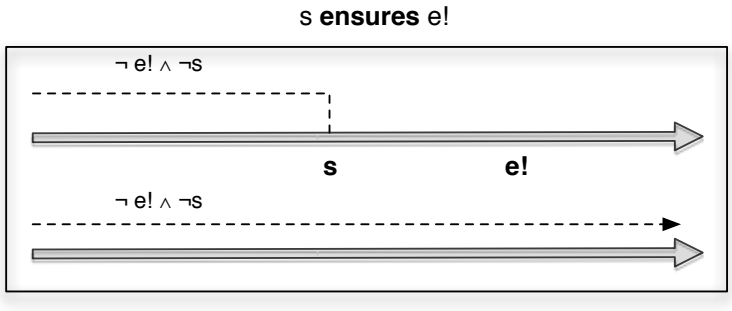


- "*s enables e?*" abbreviates "*s enables e? until false*" — once *s* (first) becomes true *e* cannot be discarded ever again and before *s* becomes true *e* cannot be executed.



- "*initiallyEnabled e?*" abbreviates "*true enables e?*" — the event *e* will never be discarded.

- "*s ensures e*!" abbreviates "*s ensures e*! *before false*" — after *s* (first) becomes true, but not before, *e* will be published.



Appendix C. Specifications of elements of the case study

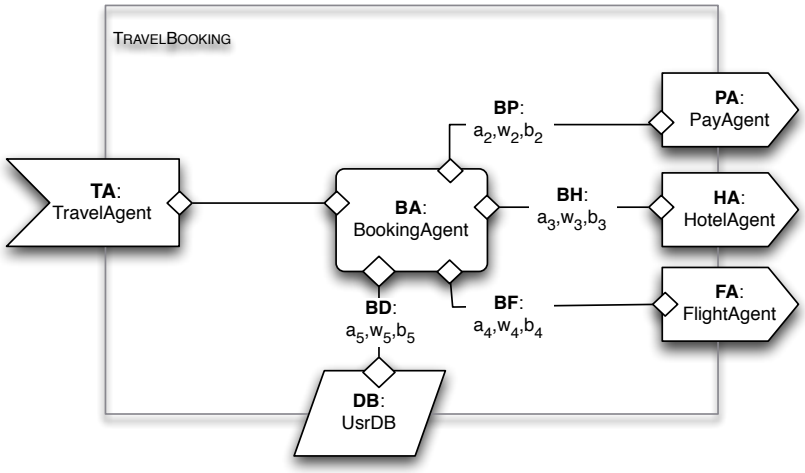


Figure C.5: The structure of the module *TravelBooking*. The service is assembled by connecting a component *BA* of type *BookingAgent* to three external service instances *PA*, *HA* and *FA* with interface types *PayAgent*, *HotelAgent* and *FlightAgent* (respectively) and the persistent component (a database of users) *DB* of type *UsrDB*. *BP*, *BH*, *BF*, and *BD* are the wires that interconnect the several parties and *TA* of type *TravelAgent* is the interface through which service requesters interact with the *TravelBooking* service.

```
BUSINESS PROTOCOL HotelAgent is
   INTERACTIONS
      r&s lockHotel
         🔔 checkin,checkout:date,
            name:usrData
         ✉ hconf:hcode
   BEHAVIOUR
      initiallyEnabled lockHotel🔔?
      lockHotel✓? enables
            lockHotel✞? until date(time)≥lockHotel.checkin
```

Figure C.6: The specification of the service interface of a *HotelAgent* written in the language of business protocols. A *HotelAgent* can be involved in one interaction named *lockHotel* that models the booking of a room in a hotel. The interaction is declared to be of type r&s, which means that it is initiated by the co-party (the component BA) which expects a reply from the party (the application that binds to the interface). Parameters are declared for the event that starts the interaction (indicated under 🔔) and the reply (indicated under ✉). Some properties of this interaction are specified: a booking request will be accepted once the service is instantiated (first formula under BEHAVIOUR) and a booking can be revoked after a commit and up until the check-in date (second formula).

```
BUSINESS PROTOCOL TravelAgent is
  INTERACTIONS
    r&s login
      🔔 usr:usrName, pwd:password
    r&s bookTrip
      🔔 from,to:airport,
         out,in:date
      ✉ fconf:fcode,
         hconf:hcode,
         amount:moneyValue
    snd payNotify
      🔔 status:bool
    snd refund
      🔔 amount:moneyValue
  BEHAVIOUR
    initiallyEnabled login🔔?
    login✉! ∧ login.reply enables
      bookTrip🔔? until time≥login.useBy
    bookTrip✓? ensures payNotify🔔!
    payNotify🔔! ∧ payNotify.status enables
      bookTrip✞? until date(time)≥dayBefore(bookTrip.out)
    bookTrip✞? ensures refund🔔!
```

Figure C.7: The specification of the provides-interface of the service module *TravelBooking* written in the language of business protocols. The service can be involved in four interactions (*login*, *bookTrip*, *payNotify* and *refund*) that model the login into the system, the booking of a trip, the sending of a receipt and refunding the client of the service (in case a booking is returned). Five properties are specified for these interactions: a login request will be accepted upon instantiation of the service; a request for booking a trip will be accepted after a positive reply to the login request until the period for submitting the request expires; a notification of payment will be sent after the booking is accepted; the booking can be revoked after the notification of payment until the day before departure; revoking the booking will lead to a refund.

```
BUSINESS ROLE BookingAgent is
INTERACTIONS
   r&s login
      🔔 usr:usrName,
         pwd:password
   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,
         payee:accountNumber,
         payService:serviceId
   s&r payment
      🔔 amount:moneyValue,
         payee:accountNumber,
         originator:usrData,
         cardNo:payData
      ✉ proof:pcode
   s&r bookHotel
      🔔 checkin:date,
         checkout:date,
         traveller:usrData
      ✉ hconf:hcode
   snd payAck
      🔔 proof:pcode,
         status:bool
   snd payNotify
      🔔 status:bool
   rcv ackRefundRcv
      🔔 amount:moneyValue
   snd refund
      🔔 amount:moneyValue
   s&r log
      🔔 usr:usrName,
         pwd:password
   s&r getData
      🔔 usr:usrName
      ✉ traveller:usrData,
         cardNo:payData
ORCHESTRATION
   local s:[START, AUTHENTICATING, LOGGED, QUERIED, FLIGHT_OK,
         QUERIED, DATA_OK, FLIGHT_OK, HOTEL_OK, CONFIRMED,
         END_PAYED, END_UNBOOKED, COMPENSATING, END_COMPENSATED]
   transition Login
      triggeredBy login🔔
      guardedBy s=START
      effects s'=AUTHENTICATING
      sends   log🔔 ∧ log.usr=login.usr  ∧ log.pwd=login.pwd
   transition AuthenticationAnswer
      triggeredBy log✉
      guardedBy s=AUTHENTICATING
      effects log.Reply ⊃ s'=LOGGED
         ∧ ¬log.Reply ⊃ s'=END_UNBOOKED
      sends   login✉ ∧ login.reply=log.reply
   transition Request
      triggeredBy bookTrip🔔
      guardedBy s=LOGGED ∧ ¬(time ≥ login.useBy)
      effects ¬(date(time) ≥ bookTrip.out) ⊃ s'=QUERIED
         ∧ (date(time) ≥ bookTrip.out) ⊃ s'=END_UNBOOKED
      sends ¬(date(time) ≥ bookTrip.out) ⊃ getData🔔
            ∧ getData.usr=login.usr
         ∧ (date(time) ≥ bookTrip.out) ⊃ bookTrip✉
            ∧ ¬bookTrip.reply
   transition DataAnswer
      triggeredBy getData✉
      guardedBy s=QUERIED
      effects s'=DATA_OK
      sends bookFlight🔔 ∧ bookFlight.from=bookTrip.from
            ∧ bookFlight.to=bookTrip.to
            ∧ bookFlight.out=bookTrip.out
            ∧ bookFlight.in=bookTrip.in
            ∧ bookFlight.traveller=getData.traveller
```

Figure C.8: Part of the specification of a *BookingAgent* using the language of business roles.

```
BUSINESS PROTOCOL FlightAgent is
  INTERACTIONS
    r&s lockFlight
      🔔 from,to:airport,
         out,in:date,
         traveller:usrData
      ✉ fconf:fcode,
         amount:moneyValue,
         payee:accountNumber,
         payService:serviceId
    rcv payAck
      🔔 proof:pcode,
         status:bool
    snd payRefund
      🔔 amount:moneyValue
  BEHAVIOUR
    initiallyEnabled lockFlight🔔?
    lockFlight✉! ∧ lockFlight.reply enables payAck🔔?
    payAck🔔? ∧ payAck.status enables
              lockFlight✞? until date(time)≥lockFlight.out
    lockFlight✞? ensures payRefund🔔!
```

Figure C.9: The business protocol followed by a *FlightAgent*. A *FlightAgent* can engage in three interactions (*lockFlight*, *payAck* and *payRefund*) and offers the following properties: a request for a flight will be accepted upon instantiation of the service; an acknowledgment of payment will be processed after (and only after) the request for the flight is accepted; once the pay acknowledgment is received, the flight booking can be revoked until the date of departure; revoked bookings will be refunded.

INTERACTION PROTOCOL RStraight.I(d_1,d_2,d_3)O(d_4)D(v:time) **is**

ROLE A
s&r S_1
🔔 i_1:d_1, i_2:d_2, i_3:d_3
✉ o_1:d_4

ROLE B
r&s R_1
🔔 i_1:d_1, i_2:d_2, i_3:d_3
✉ o_1:d_4

COORDINATION
S_1 = R_1
$S_1.i_1$=$R_1.i_1$
$S_1.i_2$=$R_1.i_2$
$S_1.i_3$=$R_1.i_3$
$S_1.o_1$=$R_1.o_1$
reliableOn S_1
reliableOn R_1
S_1 **noLaterThan** v
R_1 **noLaterThan** v

Figure C.10: An interaction protocol that connects an s&r interaction with a r&s interaction where the initiation-event (🔔) has three parameters and the reply-event (✉) has just one. According to this interaction protocol the two interactions are equivalent and their parameters are observed in the same way from the point of view of the two parties involved in the interaction. Moreover, reliability in the delivery of events associated with the two interactions is required. The delay introduced by the wire in the transmission of these events is bounded to the value v, which is a time value left undefined until the protocol is applied. Similarly, d_1, d_2, d_3 and d_4, which are the data sorts of the parameters, are also left undefined until the interaction protocol is applied.

BA BookingAgent	a_3	**BH**	b_3	**HA** HotelAgent
s&r bookHotel	S_1		R_1	**r&s** lockHotel
🔔 checkin	i_1	RStraight.	i_1	🔔 checkin
checkout	i_2	I(date,date,usrData)	i_2	checkout
traveller	i_3	O(hcode)D(60)	i_3	name
✉ hconf	o_1		o_1	✉ hconf

Figure C.11: The connector that binds the *BookingAgent* to the *HotelAgent* using a *RStraight* interaction protocol. Variables S, $S.i_1$ $S.i_2$, $S.i_3$ and $S.o_1$ are associated with the interaction name *bookHotel* and parameters *checkin*, *checkout*, *traveller* and *hconf*, respectively, in order to define a morphism from the Role A of the protocol into the signature of the *BookingAgent*. Variables R, $R.i_1$ $R.i_2$, $R.i_3$ and $R.o_1$ define a morphism from the Role B into the signature of *HotelAgent*. The datatypes of parameters *checkin*, *checkout*, *traveller* and $hconf$ (*date*, *date*, $usrData$ and *hcode*, respectively) and the time value 60 are used to instantiate the formal parameters of the interaction protocol.