Modelling Business Conversations in Service Component Architectures

João Pedro Abril de Abreu

Thesis submitted for the degree of Doctor of Philosophy at the University of Leicester

July 2009

Abstract

Modelling Business Conversations in Service Component Architectures

João Pedro Abril de Abreu

Service-oriented computing (SOC) is a new paradigm for creating and providing business services via computer-based systems. In SOC, services are computational entities that can be published together with a description of business functionality, discovered automatically and used by independent organizations to compose and provide new services. Although several technologies are being introduced with the goal of supporting SOC, the paradigm lacks theories and techniques that enable the development of reliable systems.

SENSORIA is a research project that addresses these aspects by developing mathematically-based methods for engineering service-oriented systems. Within this project, the SENSORIA Reference Modelling Language (SRML) is being developed to support the design of services at a level of abstraction that captures business functionality independently of specific technologies.

In this thesis, we provide a semantics for the fragment of SRML that supports the design of composite services from a functional point of view. The main goal of this research is to give system designers the means to design new services by integrating existing services, while making sure that the resulting system provides the intended business functionality — what is called *correctness of composition*.

In order to address this goal, we define a mathematical model of computation for service-oriented systems based on the typical business conversations that occur between the constituents of such systems. We then define the semantics of the SRML language over this model and base it on a set of specification patterns that capture common service behaviour. We show that the formality of the language can be exploited with practical gains, by proposing a methodology for model-checking the correctness of service compositions.

Our results indicate that a formal approach to service design based on the conversational nature of business interactions can promote the development of functionally correct services. Furthermore, this approach can optimize the development of service-oriented systems by allowing conceptual errors to be identified and corrected before the systems are built.

Acknowledgements

I thank José Luiz Fiadeiro for supervising all aspects of the research that led to this thesis. His constant availability, expertise and pragmatism have been essential for the outcome of this work.

I thank Stefania Gnesi and Franco Mazzanti with whom I have worked on the logic of service-oriented computing and the model-checking methodology. I thank them both for always making me feel welcome to discuss all aspects of the research and sharing their time. It was a pleasure to visit you and the rest of your team at ISTI, as it was meeting Stefania's family.

Special thanks are due to Antónia Lopes for recognizing my ability to carry this work and creating that opportunity.

I want to thank the members of the SENSORIA project, in particular Antónia and Laura Bocchi with whom I have worked directly on the SRML language.

I thank Emilio Tuosto for reviewing the thesis and for very interesting and open-minded discussions about computer science (among other subjects).

I also wish to thank all the members of the Department of Computer Science at the University of Leicester and in particular my fellow PhD students for receiving me so well during these years.

Lastly, I wish to thank my family and friends who have been essential throughout the process of doing this work, as they are in all other parts of my life. I take the opportunity to thank some of them next. Gabi, Tiago, António and especially Carla, I thank you for not being just "acquaintances" in Leicester; having met you has been one of the main outcomes of my PhD. I thank Donna for welcoming me into her group of friends: Beau, Alice, Grace, Fred, Jerry, and Vincent, with whom I have spent privileged time during my first year in Leicester; and Jose Bustos who, for six months, shared his joy with Leicester. I thank my brother Rodrigo, Pedro Rodrigues, Lara, Nuno, Vicente, Miguel, Rosa, Tiago Ruivo, Fatinha, Rui Correia, Daniele Strollo and Carla who, during these years, have each in their own way contributed actively towards my well being. Finally, I thank my mother, father and brother who keep on supporting me.

OBRIGADO.

Contents

Chapter 1

Introduction

1.1 Service-oriented computing

Service-oriented computing (SOC) is a new paradigm for creating and providing business services via computer-based information systems. In SOC, services are computational entities that can be published together with a description of business functionality, discovered automatically and used by independent organizations to compose and provide new services (or for the internal use of that organization). 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).

The particularity of SOC with respect to other software development paradigms is that services are not composed of fixed software components that are known in advance and whose coupling can be defined before the service is required; in SOC, the provision of a service may require binding to other services whose providers will be selected each time they need to be used — this is because an organization that requires a service is interested in getting the best possible quality of service from an ever-changing world of service providers (and also wants the selection process to take into account the specific context in which the service is being requested).

This dynamic and loose coupling of components sets SOC apart from other paradigms in two fundamental ways: on the one hand, because the choice of external services is done at run-time and because that choice may vary each time those services are requested, the configuration of components that provides a given service keeps changing in ways that cannot be predicted; on the other hand, because services must be made available for use by an heterogeneous universe of independent organizations, they must be described in a way that abstracts their business functionalities from the specific technologies that are used to implement them — the technical details about the way a service is coupled to its client are left to be dealt with by an independent middleware layer each time the service is requested.

While several technologies are being introduced by different stakeholders with the goal of supporting SOC, the paradigm still lacks sound foundational theories and techniques for addressing challenges such as automated composition, performance, security and safety. SENSORIA is a research project that addresses these aspects by developing formal methods for engineering service-oriented software [71]. Within SENSORIA, the SRML language is being developed to support the design and analysis of services at the "technology-agnostic" business level [36].

1.2 State of the art in service-orientation

The evolution of SOC is being driven by the "concurrent" and not always "interactive" efforts of several standardization interest groups such as W3C, OASIS or DARPA. At least three proposals have been made for standardizing the concept of SOC: one by W3C [18], which was abandoned in the year 2004, another by the OASIS consortium [52], and more recently a third one by the Open Group [42]. These proposals define what is loosely referred to as Service-Oriented Architecture (SOA). Most of the efforts that have been placed on making SOC a technological reality build upon the notions defined by SOA.

Web Services

The major approach to the implementation of SOA consists of introducing technologies that rely on the existing standards for the World Wide Web like the HTTP and SOAP protocols or XML based languages — such technologies implement what are called Web Services.

Within this approach, a main goal has been to create standards for allowing service accessibility through the web. The language WSDL [26], which is developed by W3C, has been adopted as the standard for describing the interface of web-acessible services. A WSDL description essentially declares the operations that are provided through the web by some HTTP accessible software component. A WSDL operation declaration is much like a method

signature of a standard programming language; the main difference is that WSDL is XML-based to allow independence from the programming language in which the component is implemented.

As an extension to the Web Service approach, and following the trend of the Semantic Web [13], languages are being developed for adding semantic content to web service descriptions — these technologies implement what are called Semantic Web Services. The DARPA research agency, which is part of the U.S. Department of Defense, has developed the OWL-S language, which is based on an ontology tailored for web services and allows WSDL descriptions to be extended with information about what the services do and how they should be used [55]. On this side of the Atlantic, the European Semantic Systems Initiative, which capitalizes on several EU research projects, is developing the WSML language based on the WSMO ontology for web services [35]. The purpose of both of these languages is to attach semantic information to web service descriptions that can be used to guide the process of discovery, negotiation and composition of services.

Service composition

In addressing the problem of providing new services by combining existing services — what is refered to as service composition — two different approaches have emerged: orchestration and choreography [25]. An orchestration defines the local behaviour of an "orchestrator" component (or more than one) that interacts with a set of existing services in order to a accomplish a certain business goal. A choreography is a global description of the messages that are exchanged by a set of parties to accomplish a business goal. In a sense, a choreography is a specification of the intended behaviour of a system, which is not guaranteed to be implementable [7], while an orchestration is an implementation that is not guaranteed to provide the intended behaviour. Because of this, efforts need to be made, on the one hand, to understand how (and if) a given choreography can be implemented and, on the other hand, to understand if a given orchestration generates the correct behaviour — the latter is referred to as "correctness of composition".

The two languages considered to be de facto standards for orchestration and choreography are BPEL [8] and WS-CDL [47], 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 allows orchestrating web services that are accessible through a WSDL interface. WS-CDL is a standardization candidate of W3C that can be used to specify a choreography of peers using WSDL (or XML schema) typed messages.

Service Component Architecture

Resulting from an industry collaboration, the Service Component Architecture (SCA) is a set of specifications that intend to standardize the notion as well as the technological process of service composition in a way that allows the use of several standard technologies for orchestrating web services [12]. In SCA, new services (or just applications) are built by providing an XML-based definition of a configuration of components — which can be implemented using technologies such as BPEL, C++ or JAVA — that communicate with a set of existing services described using WSDL.

Service publication and discovery

The UDDI (Universal Description, Discovery and Integration) is an XML-based protocol for creating "yellow pages" of web services [27]. A UDDI server contains a list of WSDL descriptions that (in theory) correspond to web available services. While the UDDI is considered the technological standard that supports service publication and discovery, it allows little more than manual searches for web services based on business names or textual descriptions of the services, in the style of search engines. The interest of companies in maintaining UDDI lists has been very reduced — in fact, IBM, Microsoft and SAP, have announced the discontinuation of their UDDI business registries at the end of 2005. Research indicates that, at the moment, generic search engines like Google or Yahoo! are in fact more reliable for finding active web services than UDDI registries [6]. Unfortunately, like UDDI

registries, search engines do not support the automation of service discovery.

1.3 Shortcomings and challenges

The technologies that are currently available for SOC are still far from supporting the paradigm in its full dimension. The technologies that are available for service publication, like the UDDI, offer very little support for automated discovery of services. The technologies that are available for implementing service composition, such as BPEL/WSDL, rely on static references to existing services; yet the dynamic nature of SOC requires service to be continually updated, not only to ensure the best quality of service, but also because service availability keeps changing. On the other hand, automatic service discovery and composition requires accessing rich behavioural descriptions of services, which are not supported by current web service standards like WSDL. Technologies such as those that support semantic web services are definitely a step forward towards automatic discovery and composition, but such technologies still lack the support of reliable automatic methods for matching service descriptions with business requirements.

From the engineering point of view, SOC requires the introduction of well-founded analytical methods and tools to support every stage of the service development process, from design to maintenance. It is essential to provide developers with the means to systematically create services that are reliable, functionally correct, secure and efficient. Yet, at this stage, there is little support for the development of services in more than an intuitive manner.

This is a level at which the software engineering community is making a significant contribution to SOC. Computer scientists are bringing the formal mathematical techniques that have been proved useful for software engineering to help fulfill the full potential of the SOC paradigm. A vast amount of work is being done to address issues such as composition, security or discovery using formalisms such as process algebras, automata and Petri nets. Efforts are being placed not only on giving theoretical foundations to SOC, but also on supporting the design process and guiding the use and evolution of technological standards.

1.4 The SRML approach

The SENSORIA Reference Modelling Language (SRML) is being developed within project SENSORIA as a prototype domain-specific language for modelling service-oriented systems at the business (high) level of abstraction. SRML is inspired by the Service Component Architecture (SCA) composition 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"* (p. 2 of [11]).

While both SCA and SRML are being developed 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++, 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.

Furthermore, SRML provides support for modelling not only the functional properties of services (i.e. the business logic), but also the aspects that concern the run-time discovery and binding of services [38], which in our view are at the centre of the SOC paradigm — such aspects are not addressed by SCA.

The main novelty of SRML in addressing the business logic of services is that it adopts a set of complex 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 characterized 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 (a message can be a request, a cancelation, and so on);
- service interface behaviour is specified using message correlation patterns that are typical of business conversations; and
- there are pre-defined business conversation protocols that services can follow (in particular, requester and provider protocols).

Additionally, these conversation protocols capture the crucial role that time has in the outcome of the negotiations. In the world of SOC, the deals that a service provider puts on offer keep changing as product availability changes or markets evolve. 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.

Because of the importance that time has in the business logic of services, the delays that are introduced by the infrastructures over which services run become crucial in SOC. This is why in SRML communication is asynchronous — messages are transmitted by wires, which may introduce a delay — and support for specification and analysis of service performance is also being developed [14].

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 [66] for a discussion about the benefits of declarative specifications). The formal semantics of SRML, of which a part is defined in this thesis, aims at supporting the design process with analytical techniques and tools [3, 14].

In the remaining of this section we give a brief outline of the SRML language. We put the emphasis on the part of the language that concerns

the business logic of service composition, which is the part of SRML we will focus on throughout this thesis.

1.4.1 Service composition

The design of composite services in SRML resembles component-based development (CBD) in the sense that new services result from the orchestration of independent units of behaviour. Nonetheless, SRML moves away from CBD because in SOC there is no fixed system of components that services can be programmed to draw from but, instead, an evolving universe of business functionality that service providers publish and that can be discovered and used by other services as these execute.

SRML adopts the SCA assembly model according to which new services are composed by interconnecting a set of elementary components to a set of external services [12]; the new service is provided through an interface to the resulting system. The business logic of such a composite 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 optimizing the satisfaction of service level agreement constraints [38]).

The elementary unit for specifying service 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. In addition, a module defines constraints on the process of discovery and selection of the required services.

Figure 1.1 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 com-

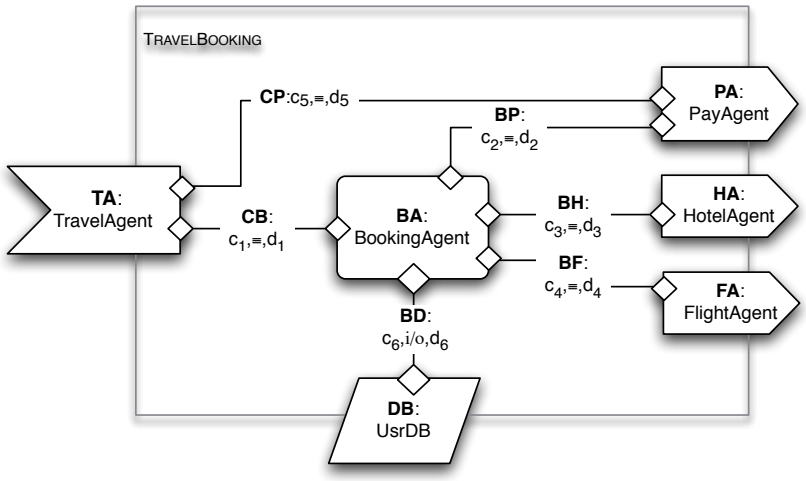


Figure 1.1: The structure of the module *TravelBooking*. The service is assembled by connecting a component *BA* of type *BookingAgent* to three external service instances *PA*, *HA* and *FA* with interface types *PayAgent*, *HotelAgent* and *FlightAgent* (respectively) and the persistent component (a database of users) *DB* of type *UsrDB*. *CB*, *CP*, *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.

ponent *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 [33]. *TA* is the interface through which service requesters interact with the *TravelBooking* service. In SRML, interactions are peer-to-peer between pairs of entities connected through wires. *CB*, *CP*, *BP*, *BH*, *BF* and *BD* are the wires in $TravelBooking$.

1.4.2 Functional behaviour and correctness

Each party (component or external service) is specified through a declaration of 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 1.1), its specification is an orchestration given in terms of state transitions — using the language of *business roles*. 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* is used for defining such specifications. External services are specified by their interface behaviour and not by their internal workflow. Figure 1.2 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 1.3 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*.

Business protocols use a set of predefined behaviour patterns that capture common service interface behaviour — this is because, in order to make on-the-fly service discovery efficient, SRML advocates that the matching of requires-interfaces of a module with provides-interfaces of potential service

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

Figure 1.2: 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. Some properties of this interaction are specified: a booking request can be made once the service is instantiated and a booking can be revoked up until the check-in date.

```
BUSINESS PROTOCOL TravelAgent is
  INTERACTIONS
    r&s login
      🔔 usr:username, pwd:password
    r&s bookTrip
      🔔 from,to:airport,
         out,in:date
      ✉ fconf:fcode,
         hconf:hcode,
         amount:moneyvalue
    snd payNotify
      🔔 status:boolean
    snd refund
      🔔 amount:moneyvalue
  BEHAVIOUR
    initiallyEnabled login🔔?
    login✉! ∧ login✉.Reply enables bookTrip🔔?
    bookTrip✓? ensures payNotify🔔!
    payNotify🔔! ∧ payNotify🔔.status enables bookTrip✞?
    bookTrip✞? ensures refund🔔!
```

Figure 1.3: The specification of the provides-interface of the sevice 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.

providers must rely on repositories of business protocols that are hierarchically organized at design-time.

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.

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

A service module is said to be functionally correct if the behaviour that is advertised in its provides-interface is entailed by the composition of components, required services and wires that the module specifies.

1.4.3 Service discovery and binding

In [38], a formal notion of service discovery and binding is given that defines how services are assembled at run-time — in broad terms, services bind to the discovered services by replacing their requires-interfaces with the provides-interfaces of the discovered modules. The semantics of this binding process is based on an algebraic composition of service modules (see [38]) that guarantees that a new service module is correct as long as the modules that are discovered and bound are correct as well.

The proposed notion of service discovery and binding is independent of the logic and languages in which the properties of the module and each of its parts is specified. While the notion of correctness of a module is expressed in terms of logical entailment, the mechanisms that SRML provides for composing different modules is independent of any such logic. The particular choice of logic operators, their semantics and proof-theory are essential for supporting the design of composite services from a functional point of view but not for

the semantics and pragmatics of the discovery and binding process that takes place at run-time.

The aspects that concern discovery and binding are addressed in SRML by a particular fragment of the language, which is outside the scope of this thesis. This fragment allows the specification of policies that define how the external services required by a module are selected and when they are bound. As a short summary, we only mention that:

- The external policy concerns the way the module relates to external parties: it declares a set of constraints that have to be taken into account during discovery and selection. Every constraint involves a set of variables that includes both local parameters of the service being provided (e.g. the percentage of the cost of a trip that is refundable) and standard configuration parameters selected from a fixed set of types availability, response time, message reliability, inter alia;
- The internal policy of a module concerns the timing of the binding of its interfaces and instantiation of its components and wires.

For more details on how discovery and binding is addressed in SRML we refer the reader to [38].

1.5 Contribution and structure of the thesis

In this thesis we focus on the fragment of SRML that supports the design of composite services from a functional point of view. More precisely, we define the semantics of the languages that SRML provides for specifying functional behaviour, which we have introduced briefly in Section 1.4, and we investigate how the verification techniques being developed within project SENSORIA — specifically the UCTL temporal logic and the UMC model-checker [1] — can be used to support the design of functionally correct service compositions.

In particular, the language of business protocols, which is used for specifying service interface behaviour (both of required services and of modules

[1]The UCTL logic and the UMC model-checker (which is based on UCTL), are being developed within SENSORIA by *ISTI-CNR* in Pisa.

themselves), is defined over UCTL so that the mechanism of module composition (discussed in [37] and [38]) can work — this is because, the business logic of a module relies on the discovery of services (provided by other modules) that satisfy a set of specified properties.

In order to give meaning to the languages presented in Section 1.4 we define a model of computation that captures the conversational nature of the interactions that occur in SOC. This model together with the model of service discovery and binding (presented in [38]) provide the mathematical characterization of service-oriented computation that is required by SRML for developing formal methodologies for engineering service-oriented systems.

The following is a short summary of each of the contributions made by this thesis.

Semantic domain (Chapter 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. In particular, we define the typical requester and provider conversation protocols. This model constitutes the semantic domain of SRML.

Logic of SOC (Chapter 4) We give a logical characterization of our model of computation, using the logic UCTL. That is, we capture the properties that characterize computation in service-oriented systems and the requester and provider conversation protocols using temporal formulas. An axiomatization is given whose soundness is proved in Appendix A.

Specification languages (Chapter 5) We formalize the languages of business roles, business protocols and interactions protocols, which are used in service modules to specify the behaviour of components, service interfaces and wires, respectively. For each of these languages, we provide a formal semantics over our model of computation.

Behaviour patterns (Chapter 5) We formalize the behaviour patterns that can be used for specifying service interface behaviour in terms of abbreviations of UCTL formulas.

Model-checking service composition (Chapter 6) We propose a methodology for supporting the development of functionally correct service modules using the UML state machine based model-checker UMC. In order to formalize our methodology, we make a formal review of the UMC model-checker.

Throughout the thesis, we rely on the *TravelBooking* case study to illustrate our framework. In Appendix B, we present the complete SRML specification of the *TravelBooking* service module. In Appendix C, we present the complete encoding of *TravelBooking* with UML state machines for the purpose of model-checking with UMC.

Most of the work that is presented in this thesis has been discussed in previous publications. In particular, the model of computation that we propose for SOC has been presented in [2]; the language of interaction protocols that is used for specifying wires is discussed in [1]; the semantics of the patterns of service behaviour and the model-checking approach that we propose for service composition is presented in [3]; and an overview of the SRML approach to service-oriented modelling is given in [39]. This thesis presents a comprehensive and unified view of that work.

Chapter 2

Related work

2.1 Formal approaches to service composition and analysis

Different formalisms and techniques are being brought to bear the challenges raised by SOC. Among the approaches that address service composition some focus directly on supporting the use of existing technological standards, while others (like SRML) remain largely "technology agnostic". Among the formalisms that are being used for defining service composition, process algebras, Petri nets and automata are becoming pervasive. The analysis techniques that are associated with these formalisms, such as model-checking with temporal logic and bisimulation (in the case of process algebras), are being used extensively for supporting the correct development of service compositions.

In this section we situate the work that is presented in this thesis by discussing some of the proposals found in the literature for addressing the problem of service composition and analysis. While we have organized the discussion by formalism (process algebra, Petri nets, automata and temporal logic), it is beyond our objectives to define a taxonomy of approaches. The approaches that we have selected are related to our work either because they have (or at the surface seem to have) similar goals or because they propose similar modelling or analysis techniques.

2.1.1 Process algebras

Process algebras (or calculi) are a family of languages that allow precise descriptions of concurrent computations. Several prototype programming languages for concurrent systems have been proposed that have a process algebra based semantics (e.g. [62, 68]), some of which promote component-based development (e.g. [5])

One of the main characteristics of process algebras is that they rely only on very few syntactic primitives. This in turn, makes it possible for formal techniques to be used for manipulating and reasoning about processes using techniques such has bisimulation. The disadvantage of having a very

simple syntax is that it becomes hard to model domain specific phenomena like the complex conversations that characterize SOC. Because of this, most of the attempts at applying process algebras to SOC consist of extending the capabilities of well-established process algebras (typically the π-calculus [57]) with primitives that capture SOC phenomena — the following are some examples.

COWS (Calculus for Orchestration of Web Services) is a process algebra, developed within project SENSORIA, that provides operators inspired by BPEL [50]. 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 the UCTL logic and its associated model checker [34]. The strategy for model-checking COWS specifications that is proposed by the authors 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 the SRML language — are not part of COWS.

Other approaches exist that focus on abstracting subsets of BPEL with process algebras in order to analyse phenomena such as data-independent functionality [70] or fault handling [51].

Also within SENSORIA, Boreale et al. propose SCC (Service Centered Calculus) [19], 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 [58]. SCC adopts a notion of service session that allows modelling client-server relationships. SCC also provides a primitive for closing a session, which allows modelling service interruption and session termination. ORC itself was originally a calculus for modelling orchestrations, but has evolved into a full-fledged programming language for implementing orchestrations [49]. The notion of session type (see [45]) is also being brought forward to the realm of SOC to track the types of the values exchanged in each session (e.g. [4, 21]).

Vieira et al. [69] 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 characterized by the events that can happen associated with those interactions and the temporal correlation between those events.

The previous are just examples (mostly taken from project SENSORIA) of process algebras tailored for SOC. Several other processa algebras exists that, for the sake of conciseness, we do not discuss (e.g. [24], which is based on W3C's WS-CDL, or [20], which focuses on multi-party sessions).

2.1.2 Petri nets

Petri nets are a very well established formalism for modelling distributed systems (see for example [63] or [29] for an overview on Petri nets). In particular Petri nets lend themselves to modelling a class of workflows — referred to as synchronization workflows [48] — that contain the branching and synchronization syntatic primitives found in BPEL [61]. Therefore, it is not surprising that one of the main contributions of the Petri net community towards the development of SOC has been to provide a formal semantics for BPEL (e.g. [61, 44, 54]). Nevertheless, the use of Petri nets in SOC is also being studied independently of BPEL.

Narayaman et al. provide support to the semantic web service technology by defining a Petri net based semantics for the subset of OWL-S that deals with processes [59]. The authors 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 composition of services behaves globally as intended.

Yi and Kochut propose an approach in which service interface conversations are modelled by WSDL tailored Petri nets [72]. 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 composition 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 composition; the traditional algebraic techniques like bisimulation and structural equivalence can then be used to analyse service compositions [41].

2.1.3 Automata

Automata (also known as State Machines) provide a simple and intuitive way of modelling computational systems. For example, the Specification and Description Language (SDL) is a standard for specifying telecommunication systems in an object-oriented fashion that is based on state machines [65]. SDL is suited for the specification of generic real-time systems where several concurrent activities communicate using discrete signals and is a good illustration of the power of state machine based formalisms for specifying distributed systems. Several attempts at harnessing this power for the purposes of SOC are being pursued.

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 [40] — in particular, the authors propose a tool supported method for verifying the correctness of BPEL compositions 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).

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 [30]. Dong et al. define a timed automata based semantics for ORC, which they proof to be equivalent to the official semantics of ORC [31], and advocate the use of timed automata based model-checkers for verifying the correctness of ORC orchestrations.

Beek et al. discuss the use of UML state machines, the logic UCTL, and the model checker UMC [56] in the design of an extension of the SOAP communication protocol tailored for service-oriented computing [9].

2.1.4 Temporal logic

Temporal logic has been extensively used for specifying properties of concurrent and distributed systems independently of the processes that guarantee those properties. Typically, temporal logic is 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. [50]), Petri nets or automata (e.g. [9]). In SRML, we use temporal logic not only to express the properties that are expected of a composite service, but also to model the external services that are required by the composition. In particular, SRML makes use of a set of patterns of temporal logic that capture commonly occuring requirements in business interactions. The use of patterns of logic for modelling system requirements has been addressed in the past (e.g. [53, 32]). 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 [66, 67]. 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.

2.2 Where SRML stands

There is a multitude of formal languages intended for supporting the development of functionally correct service compositions. A fundamental difference between SRML and those languages 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. These constructs can be used directly to model service behaviour — specifically SRML contains:

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

As far as we know, the existing approaches (some of which have been discussed in this chapter) support the modelling of business conversations in a much more generic way. For example, some calculi support conversations only in the sense that the set of messages can be partitioned into conversations (e.g. [50, 69]). Other approaches use generic workflow descriptions to model conversational protocols of services (e.g. [72]). To the best of our knowledge, at the moment there is no other approach in which the business nature of service conversations is captured in such detail as in SRML.

Another particularity of SRML, derives from the fact that SRML is inspired by the Service Component Architecture. SRML uses two different types of languages for specifying the components of a service: an automata-based language is used for specifying the internal components that are known a priori and a more abstract logic-based language is used for specifying the services that need to be discovered. This is because, in SOC, services must be discovered based on their conversational protocols and not on the workflows

that they follow. Most approaches model services as processes which, in our opinion, does not reflect the true nature of service discovery.

In fact, when we compare SRML to other approaches to service composition, it stands out that SRML places itself at a higher level of abstraction. This is because 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 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 has been advocated by van der Aalst and Pesic for modelling services [66] — nonetheless the authors have focused their efforts on monitoring the workflow of services at run-time and have not addressed the issue of service design. With SRML, we adopt the same philosophy and 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 that business logic is implemented.

Chapter 3

A semantic domain for service-oriented computing

In this chapter we present the semantic domain that underlies our framework. In summary, we see service-oriented systems as networks of computational entities that interact by exchanging messages whose types are meaningful from a business point of view. We model the behaviour of such systems using transition systems in which the transitions represent the exchange and processing of messages.

The chapter is organized as follows: Section 3.2 defines the notion of configuration (i.e. network of interacting nodes); Section 3.3 defines which events (i.e. messages) can be exchanged between the nodes of a configuration; Section 3.4 defines how a configuration can compute — in particular it defines how events are transmitted asynchronously between nodes and how the computation performed by a configuration can be modelled with a transition system; Section 3.5 defines the prototypical requester and provider conversation protocols.

3.1 Data modelling

Our approach focuses on the patterns of message exchange that are typical in SOC. Therefore, throughout this thesis, we will abstract from the data modelling aspects by considering a fixed data signature

$$\Sigma = \langle D, F \rangle$$

where D is a set of data sorts (such as *int*, *boolean* and so on) and F is a $D^* \times D$-indexed family of sets of operations over the sorts (such as addition, conjunction, and so on). We further assume that $time, boolean \in D$ are datatypes that represent the usual concepts of time and truth values, and that the usual operations over time and truth values are in F. We assume a fixed algebra

$$\mathcal{U}$$

for interpreting Σ.

3.2 Configurations

A *configuration* is a graph where the nodes are entities capable of performing computations and the edges are wires that connect those entities. The graph does not have multiple edges, meaning that for every two nodes there is either a single wire connecting them or they are not directly connected. The graph does not have loops either, meaning that a node cannot be connected to itself. The graph is undirected because wires do not have a direction associated with them; wires are able to transmit messages both ways as we will see further ahead. We further distinguish between nodes that are able to perform parallel computations and those that can only perform sequential computations.

Definition 3.2.1 (Configuration) *A* configuration *is a triple*

$$\langle N, WIRE, PLL \rangle$$

such that:

- $\langle N, WIRE \rangle$ *is a simple graph (undirected, without self-loops or multiple edges) where* N *is a set of computational nodes and the relation* $WIRE \subseteq N \times N$ *is the set of edges (the wires that connect the nodes);*
- $PLL \subseteq N$ *are "distributed" nodes, i.e. nodes that can perform parallel computations;* $N \setminus PLL$ *are "sequential" nodes, i.e. nodes that can only perform sequential computations;*

As discussed in Section 1.1, the configurations of service-oriented systems evolve at run-time, when external services are bound to satisfy business requirements. For the purpose of this thesis, which is to model and analyse the functional properties of composite services, we restrict ourselves to static configurations, i.e. we do not model or analyse how these configurations come about (we refer the reader to [38] for an account of how the run-time reconfiguration of systems is handled in SRML).

A *service-oriented configuration (SO-configuration)* is a configuration in which the nodes interact using a particular type of (business) interactions.

In a SO-configuration the set of interactions that can take place between each pair of nodes is fixed. We distinguish between one-way and two-way interactions and assign a direction to each interaction that defines which messages can be sent and received by which nodes. As discussed in Section 1.4, messages are propagated through wires asynchronously, i.e. there is a time gap between the instant a party sends a message and the instant the message is delivered to the co-party by the wire. Associated with each wire there is a particular delay.

Definition 3.2.2 (Service-Oriented Configuration) *A* Service-Oriented configuration *is a tuple*

$$\langle N, \mathit{WIRE}, \mathit{PLL}, \Psi, \mathit{2WAY}, \mathit{1WAY} \rangle$$

where:

- $\langle N, \mathit{WIRE}, \mathit{PLL} \rangle$ *is a configuration;*
- Ψ *assigns to every* $w \in \mathit{WIRE}$ *an element* $w.\mathit{delay}^{\Psi} \in \mathit{time}_{\mathcal{U}}$ *that denotes the delay associated with wire* w*.*
- *2WAY and 1WAY are* $N \times N$*-indexed families of mutually disjoint sets of "two-way" and "one-way" interactions, respectively; we use INT to refer to* $\mathit{2WAY} \cup \mathit{1WAY}$*;*
- *For every* $n, n' \in N$*, if* $\langle n, n' \rangle \notin \mathit{WIRE}$ *then* $\mathit{INT}_{\langle n,n' \rangle} = \emptyset$*, i.e. there are no interactions between nodes that are not connected by a wire.*

Naturally, if there is an interaction between n and n', then there needs to be a wire between these two nodes for the interaction to take place; this is captured by the last condition of the definition. We will see further ahead that if interaction i belongs to $_{\langle n,n' \rangle}$ this means that i is initiated by node n.

In the next two sections we define which events can occur when a configuration computes and we define the notions of computation state, computation step and computation model for a configuration.

Throughout the rest of the Chapter we consider a fixed configuration

$$\Xi = \langle N, \mathit{WIRE}, \mathit{PLL}, \Psi, \mathit{2WAY}, \mathit{1WAY} \rangle$$

over which all definitions are given.

3.3 Interactions, events and pledges

In service-oriented computation, typical interactions are of a conversational kind that involves a durative asynchronous exchange of correlated events (messages). In SRML, two-way interactions capture a pattern of dialogue that is prevalent in service-oriented computation: a party sends a request to a co-party that replies either positively by making a pledge to deliver a property (i.e. it gives some kind of guarantee) or negatively, in which case the interaction ends; if the answer is positive the party that made the request can commit by accepting the pledge or cancel the request. If and after the requester commits, a revoke may be available that compensates for the effects of the pledge. One-way interactions capture situations in which a party sends a single message and does not expect a reply from the co-party. This type of interaction has only this one event associated with it.

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 an 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.

Associated with every one-way interaction a there is one and only one event, a♠. Every two-way interaction a has associated with it the set of five events $\{a$♠$, a$✉$, a$✓$, a$✗$, a$↟$\}$. The possible patterns of two-way interaction supported by SRML in the case in which the reply is positive are depicted in Figure 3.1.

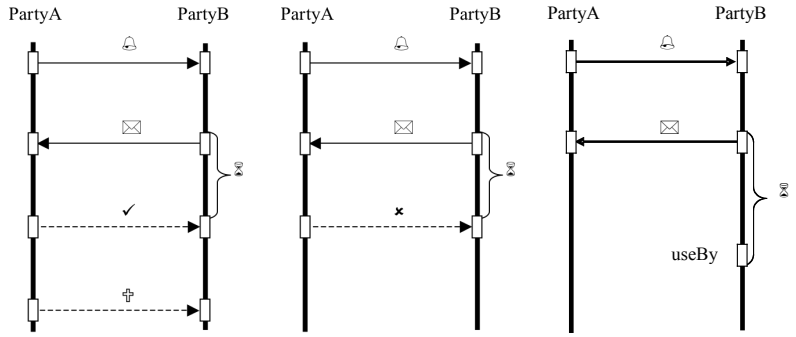


Figure 3.1: The patterns of two-way interactions that involve a positive reply. By *useBy* we denote the time point (deadline) after which *PartyB* no longer ensures the pledge. In the case on the left, the initiator commits to the pledge; a revoke may occur later on, compensating the effects of the commit-event. In the middle, there is a cancellation; in this situation, a revoke is not available. In the case on the right, the deadline occurs without a commit or cancel having occurred.

Each event has a direction associated with it; an event is sent by one party to a co-party that receives it. For every two-way interaction $a \in 2WAY_{\langle n,n' \rangle}$, the events a🔔, a✓, a✗ and a✞ can be sent by node n and received by node n', while the event a✉ can be sent by n' and received by n. More precisely:

Definition 3.3.1 (Events) *For every interaction $a \in$ and node $n \in N$, the set $E_n(a)$ of events associated with a that can be received by n is defined as follows:*

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

$$E_n(a) = \{a✉\}$$
$$E_{n'}(a) = \{a🔔, a✓, a✗, a✞\}$$
$$E_{n''}(a) = \emptyset \text{ for any other } n'' \in N$$

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

$$E_n(a) = \emptyset$$
$$E_{n'}(a) = \{a🔔\}$$
$$E_{n''}(a) = \emptyset \text{ for any other } n'' \in N$$

We also define the following sets:

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

We see E *as a WIRE-indexed or a* N*-indexed family of sets when convenient.*

Every event may have a set of parameters. In particular, every reply-event (of a two-way interaction) has at least two parameters: *Reply* defines if the reply is positive or negative and *useBy* defines an expiration time until when the pledge holds. In order to be able to refer to a parameter p without being ambiguous about which interaction that parameter is associated with, we assume that all the parameters associated with some interaction a (i.e. the parameters of the events of a) have different names. For example, if a is a two-way interaction, $a.Reply$ unambiguously denotes a parameter of $a\bowtie$ — this is because there cannot be another parameter *Reply* associated with interaction a.

Definition 3.3.2 (Parameters) *A parameter-assigning function PP assigns a D-indexed family of disjoint sets of parameters to each event in* E*, such that:*

- *For every* $a \in, e, e' \in E(a)$, $P \in PP(e)$ *and* $P' \in PP(e')$, P *and* P' *are disjoint.*
- *For every* $a \in 2WAY$, $Reply \in PP(a\bowtie)_{boolean}$ *and* $useBy \in PP(a\bowtie)_{time}$

Throughout this thesis, we assume that for every configuration Ξ there is a fixed parameter-assigning function denoted by PP^{Ξ} (or simply PP, if no ambiguity arises).

As mentioned, associated with each two-way interaction there is a pledge, which is a property (or set of properties) that holds after a positive reply. For example, a typical flight agent states a price for every requested flight that is guaranteed to hold for a period of time (i.e. until the deadline expires). In this case, the pledge is that the service requester will pay the stated price, if it commits. For every interaction a, we use $a.pledge$ to denote the pledge that is associated with that interaction.

3.4 Computation states, steps and models

In this section we define a discrete model of computation for service-oriented configurations. In this model we assume that each node is capable of publishing [1] and processing received events. We also assume that nodes are independent units of computation running in parallel and therefore can send or process events simultaneously. In our model events are transmitted asynchronously by the wires, i.e. while an event is not delivered to its destination (after being published), nodes and wires continue publishing, delivering and processing events. Once an event is delivered, it is buffered in the destination node until that node is ready to process it.

When a configuration computes, it goes through a sequence of states that are characterized, among other things, by which events are pending in wires and which events are buffered in the nodes of the configuration. The state of the system is also characterized by a time instant, the set of pledges that hold in that state and the history of events and parameters of events (i.e. a record of which events have been published, delivered, etc).

Definition 3.4.1 (Computation state) *A computation state is a tuple*

$$\langle PND, INV, TIME, PLG, HST, \Pi \rangle$$

where:

[1] We use the term "publish", which is not typically used for peer-to-peer interactions, 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 output.

- *$PND \subseteq E$ is the set of events pending in that state, i.e. the events that are waiting to be delivered by the corresponding wire;*
- *$INV \subseteq E$ is the set of events invoked in that state, i.e the events that have been delivered and are waiting to be processed;*
- *$TIME \in time_{\mathcal{U}}$ is the time in that state;*
- *$PLG \subseteq \{a.pledge : a \in 2WAY\}$ is the set of pledges that hold in that state;*
- *HST consists of four subsets of E, $HST!$,$HST¡$,$HST?$ and $HST¿$ that keep the history of event propagation; they contain the events that have been published, delivered, executed and discarded, respectively;*
- Π *assigns to each parameter $p \in PP(e)_d$ of datatype $d \in D$, with $e \in E(a)$ and $a \in$, a value $a.p^{\Pi} \in d_{\mathcal{U}}$, i.e. Π keeps the value of each parameter.*

If $s = \langle PND, INV, TIME, PLG, HST, \Pi \rangle$ is a computation state we use PND^s, INV^s, $TIME^s$, PLG^s, HST^s and Π^s to refer to the components that state.

At this point, it is important to clarify that we introduce real time in our model of computation in order to be able to formalize the notion of deadline that is associated with two-way interactions, as well as the notion of delay associated with wires. Nonetheless, it is beyond the aim of this thesis to provide a framework for modelling real-time properties — this kind of property falls within the realm of what is called *quality of service* and in this thesis we focus strictly on the functional properties of services. We refer the reader to [14] for an approach to modelling and analysing real-time performance of service-oriented systems using the SRML language.

The state of the system evolves through *computation steps*. In each computation step events can be *published* (i.e. placed in the wires by the nodes), *delivered* (i.e. taken from the wires and buffered in the nodes) or *processed* (i.e. taken from the buffers) in which case the life cycle of the event ends. We

distinguish between events that are *executed* upon processing (which typically have side effects) and those that are simply *discarded*. Nodes that do not perform parallel computation, i.e. sequential nodes, can only process one event during each step. In our model wires may not be *reliable* in the sense that they may lose events without actually delivering them to the nodes. More precisely:

Definition 3.4.2 (Computation step) *A* computation step *is a tuple*

$$\langle SRC, TRG, DLV, PRC, EXC, \Theta \rangle$$

where:

- *SRC and TRG are the source and target states;*
- *$DLV \subseteq PND^{SRC}$ is the set of events that are selected for delivery during that step;*
- *PRC is a partial function that selects for each node n such that INV_n^{SRC} is non-empty, a subset of INV_n^{SRC}, such that if $|PRC(n)| > 1$ then $n \in PLL$, i.e. PRC selects which events will be processed during that step such that only one event can be processed by each sequential node;*
- *$EXC \subseteq PRC$ is the set of events that are executed during that step. $DSC = PRC \setminus EXC$ is the set of events that are discarded, i.e. the events that are processed but are not executed;*
- *$PND^{TRG} = (PND^{SRC} \setminus DLV) \uplus PUB$ where $PUB \subseteq E$, i.e. the events that were selected for delivery will no longer be pending in the target state; the new events that become pending in the target state are those that are published during that computation step;*
- *There is a set of actually-delivered events $ADLV \subseteq DLV$ such that for every $n \in N$:*
  - *If $PRC(n)$ is defined then $INV_n^{TRG} = (INV_n^{SRC} \setminus \{PRC(n)\}) \cup ADLV_n$*

  - *If* $PRC(n)$ *is undefined then* $INV_n^{TRG} = INV_n^{SRC} \cup ADLV_n$

  i.e. the events that were processed will no longer be waiting in the target state; the events that are actually delivered to a component will have to wait until they are processed;

- Θ *assigns to each parameter in* $PP(e)_d$ *such that* $e \in PUB$ *and* $d \in D$ *(is the datatype of the parameter), an element in* $d_{\mathcal{U}}$*, i.e. the value of the parameter;*

- $TIME^{SRC} < TIME^{TRG}$*, i.e. time moves forward;*

- *SRC and TRG are such that:*

  - $HST!^{TRG} = HST!^{SRC} \cup PUB$
  - $HST¡^{TRG} = HST¡^{SRC} \cup ADLV$
  - $HST?^{TRG} = HST?^{SRC} \cup EXC$
  - $HST¿^{TRG} = HST¿^{SRC} \cup DSC$
  - $\Pi(e)^{TRG} = \Theta(e)$ *for each* $e \in PUB$
  - $\Pi(e)^{TRG} = \Pi(e)^{SRC}$ *for each* $e \notin PUB$

To summarize (and help digest), the definition says that the set of events that are pending in wires is updated during each computation step by adding the events that each node publishes — PUB — and removing the events that the wire delivers — DLV — during that step. The set of events that are waiting to be processed in each node is updated in each step by adding the events that are actually delivered — $ADLV$ — to that node and removing the events that have been processed, which are given by function PRC. The history of events is updated in each step by adding to the corresponding subsets of HST the events that have been published, delivered, executed and discarded. Events are published with a set of parameter values given by Θ, which is stored in the target state by Π. Figure 3.2 illustrates the event flow during a computation step.

A model of computation for a configuration is a transition system in which each state is labelled with a computation state and each transition is labelled

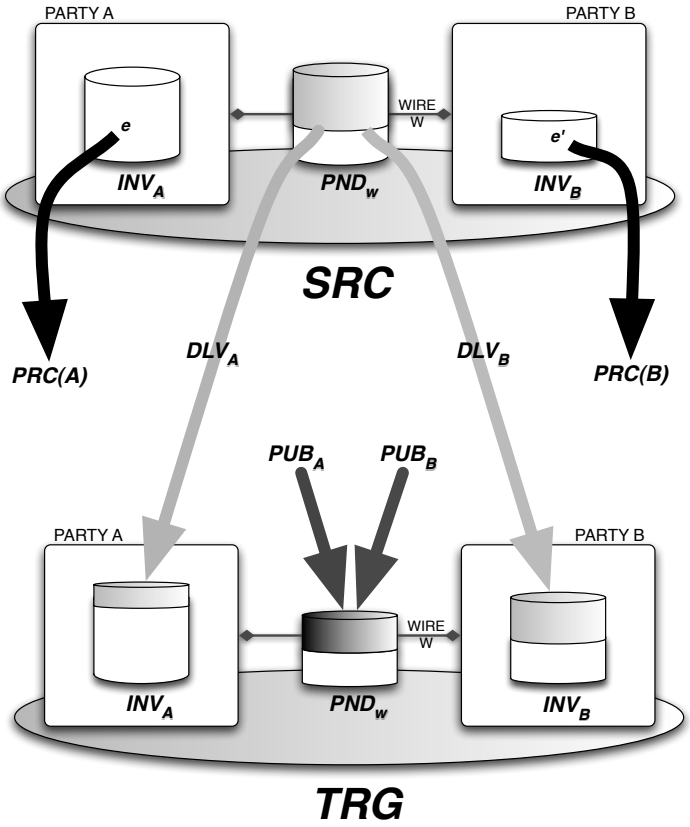


Figure 3.2: A graphical representation of the event flow during a computation step from the point of view of a wire w between a pair of nodes A and B. The system evolves from state SRC to state TRG during the step. The set of events that are published by the two nodes during the step is given by PUB_A 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 node A and enter the set INV_A while others are delivered to node 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 ($e \subseteq PRC(A)$ and $e' \subseteq PRC(B)$) and therefore are not present in the target state.

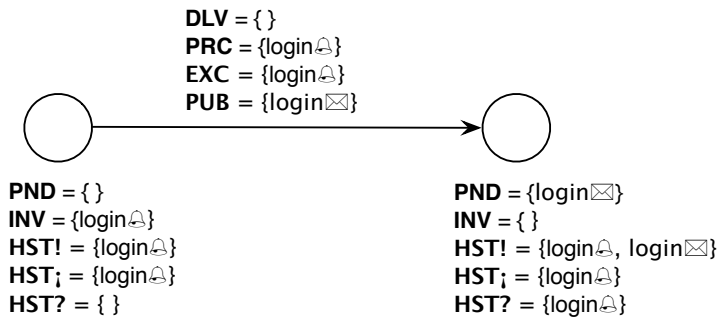


Figure 3.3: A possible transition in a model of computation (i.e. a SO-TS) for *TravelBooking*. Only a part of the labels is shown. During the transition the request-event *login*♠, which was in the set *INV* (of events waiting to be processed), is processed and executed. The reply-event *login*✉ is published and becomes pending (i.e. it is added to *PND*). The history of event propagation, represented by the family of sets *HST*, is updated in the target state.

with a computation step — we call these *Service-Oriented Transition Systems (SO-TS)* (see Def. 3.4.4). A SO-TS defines the different choices that the configuration can make while computing. Each path in the transition system represents a possible computation in terms of the order and time at which events are published, delivered and processed. Figure 3.3 illustrates by showing what could be a transition in a model of computation for *TravelBooking* (see also Figures 1.1 and 1.3).

Since we are not interested in modelling the aspects that concern error management in service-oriented systems, we restrict ourselves to those models in which every wire is *reliable*. Furthermore, we assume that the events that are pending in wires or invoked in the nodes will eventually be delivered or processed, respectively — this is what we call *fairness*. We also restrict ourselves to modelling *sessions*, which in our definition are computations during which events are not published more than once. In order to enforce this property, we only consider transition systems without cycles, i.e. transition systems where every path passes in each state at most once. In formal words, we use directed acyclic graphs (also known as DAGs) as models.

To help us define the notion of SO-TS, we first define a notion of reachability between the vertices and edges of a directed acyclic graph.

Definition 3.4.3 (Reachability) *Let $\langle S, \rightarrow \rangle$ be a directed acyclic graph, where S is the set of vertices and $\rightarrow \subseteq S \times S$ is the set of edges. The partial order relation $<$ on the set $S \cup \rightarrow$ is defined as follows, for every $s, s' \in S$ and $r = (s_1, s_2), r' = (s'_1, s'_2) \in \rightarrow$:*

- *$s < s'$ iff $s \neq s'$ and there is a directed path from s to s'*
- *$s < r$ iff $s \leq s_1$*
- *$r < s$ iff $s_2 \leq s$*
- *$r < r'$ iff $s_2 \leq s'_1$*

where $s \leq s'$ denotes $s < s' \vee s = s'$.

Definition 3.4.4 (Service-oriented transition system) *A* Service-Oriented Transition System (SO-TS) *is a tuple*

$$\langle S, \rightarrow, s_0, G \rangle$$

where:

- *$\langle S, \rightarrow \rangle$ is a directed acyclic graph, where S is the set of vertices (the states) and $\rightarrow \subseteq S \times S$ is the set of edges (the transitions between states);*
- *$s_0 \in S$ is the initial state;*
- *G is a labelling function that assigns a computation state to every state $s \in S$ and a computation step to every transition $s \rightarrow s'$, such that $G(s \rightarrow s') = \langle G(s), G(s'), \_, \_, \_ \rangle$, i.e. the source and target computation states associated with a transition are the ones that label the states of that transition.*

We use the following notation to refer to the components of the labels:

> *If s is a state ($s \in S$) we use PND^s, INV^s, $TIME^s$, PLG^s, HST^s and Π^s to refer to the components of computation state $G(s)$ (in accordance with the names used in definition 3.4.1);*

If r is a transition ($r \in R$) we use SRC^r, TRG^r, DLV^r, PRC^r, EXC^r, DSC^r and PUB^r to refer to the components of the computation step $G(r)$ (in accordance with the names used in definition 3.4.2).

Furthermore, the following properties hold for a SO-TS:

Wire Reliability *For every transition $r \in \longrightarrow$ and $w \in WIRE$, $DLV^r_w = ADLV^r_w$, i.e. wires do not lose events — we say that each wire w is reliable;*

Session *For every two transitions $r, r' \in \longrightarrow$ such that $r < r'$, $PUB^r \cap PUB^{r'} = \emptyset$, i.e. events are not published more than once on each path; and $PND^{s_0} = INV^{s_0} = \emptyset$, i.e. there are no events pending or buffered in the initial state — we say that the transition system models a session;*

Fairness *For every state $s \in S$ and event $e \in E$:*

- *If $e \in PND^s$ then there is $r \in \longrightarrow$ such that $s < r$ and $e \in DLV^r$, i.e. every event that is pending in a wire will eventually be delivered;*
- *If $e \in INV^s$ then there is $r \in \longrightarrow$ such that $s < r$ and $e \in PRC^r$, i.e. every event that is invoked (i.e. buffered in a node) will eventually be processed;*

Wire Delay *For every $w \in WIRE$, if there is a transition $r = s_1 \rightarrow s_2$ such that $e \in PUB^r$ for some event $e \in E_w$, then there is another transition $r' = s_3 \rightarrow s_4$ such that $r < r'$, $e \in DLV^{r'}$ and $TIME^{s_4} < TIME^{s_2} + w.delay^{\Psi}$, i.e. events that are published through wire w are delivered within the delay associated with w.*

3.5 The requester and provider protocols

As discussed in the beginning of the chapter, a configuration defines which interactions can take place between which nodes; in particular it defines

which events each node can publish and which events it can receive. In the case of a two-way interaction, a participating node is defined in the configuration either as the requester, meaning 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 remaining events. Although a configuration defines which events a requester and a provider can publish or process, it does not say anything about when (or if) those events will indeed be published or processed by them. The actual behaviour of a node in a two-way interaction, which is modelled by a SO-TS, can be classified according to the conditions under which the node publishes, executes or discards the events of that interaction.

In Section 3.3 we have briefly mentioned the prototypical conversation pattern that has motivated the event types that are used in our semantic model. To recapitulate, a requester sends a request to a provider that replies either positively by making a pledge to deliver a set of properties (i.e. it gives some kind of guarantee) or negatively, in which case the interaction ends; if the answer is positive the requester can commit by accepting the pledge or cancel the interaction. If and after the requester commits, the provider may accept a revoke that compensates for the effects of committing. That conversation pattern is illustrated in Figure 3.1.

That conversation can be observed when one of the nodes behaves as a requester and the other node behaves as a provider. Nonetheless, the protocol that each node follows is local and independent of the behaviour of the other nodes. Next we define the requester and the provider protocols.

We consider a fixed SO-TS

$$\langle S, \rightarrow, s_0, G \rangle$$

as model of computation for configuration Ξ.

Definition 3.5.1 (Requester) *A node $n \in N$ is said to* behave as a requester *in an interaction $a \in 2WAY_{\langle n,n' \rangle}$ iff for every transition $r = s \longrightarrow s'$*

the following properties hold:

1. *If* $a\boxtimes \in EXC^r$ *then there is* r' *with* $r' < r$ *such that* $a\spadesuit \in PUB^{r'}$
   i.e. the reply-event will not be executed before the initiation-event is published;

2. *If* $a\spadesuit \in PUB^r$ *then there is no* r' *with* $r < r'$ *such that* $a\boxtimes \in DSC^{r'}$
   i.e. the reply-event will not be discarded after the initiation event was published;

3. *If* $a\checkmark \in PUB^r$ *then there is* $r' < r$ *such that* $a\boxtimes \in EXC^{r'}$ *and* $a.reply^{\Theta^{r'}} = true$
   i.e. the commit-event will only be published after a positive reply-event was executed;

4. *If* $a\checkmark \in PUB^r$ *then there is no* $r' < r$ *such that* $a\boldsymbol{\times} \in PUB^{r'}$
   i.e. the commit-event will only be published if the cancel-event has not been published before;

5. *If* $a\boldsymbol{\times} \in PUB^r$ *then there is* $r' < r$ *such that* $a\boxtimes \in EXC^{r'}$ *and* $a.reply^{\Theta^{r'}} = true$
   i.e. the cancel-event will only be published after a positive reply-event was executed;

6. *If* $a\boldsymbol{\times} \in PUB^r$ *then there is no* $r' < r$ *such that* $a\checkmark \in PUB^{r'}$
   i.e. the cancel-event will only be published if the commit-event has not been published before;

7. *If* $a\maltese \in PUB^r$ *then there is* $r' < r$ *such that* $a\checkmark \in PUB^{r'}$
   i.e. the revoke-event will only be published after the commit-event was published.

To summarize, a requester will be ready to execute the reply to its request, but only after this request is done — this is captured by clauses 1 and 2 of the definition. A requester will only choose between cancelling or committing to the deal offered by the provider, if a deal was in fact offered — this is captured

by clauses 3 and 4. Finally, a requester will not try to revoke a deal to which it did not previously commit — this is captured by clause 5.

The provider protocol is defined next:

Definition 3.5.2 (Provider) *A node $n \in N$ is said to* behave as a provider *in an interaction $a \in 2WAY_{\langle n',n \rangle}$ iff for every transition $r = s \longrightarrow s'$ the following properties hold:*

1. *If $a\boxtimes \in PUB^r$ then there is $r' < r$ such that $a\spadesuit \in EXC^{r'}$, i.e. the reply-event will only be published after the initiation-event is executed;*
2. *If $a\spadesuit \in EXC^r$ then there is $r < r'$ such that $a\boxtimes \in PUB^{r'}$, i.e. a reply-event will be published after the initiation-event is executed;*
3. *If $a\checkmark \in EXC^r$ then there is $r' < r$ such that $a\boxtimes \in PUB^{r'}$, $a.reply^{\Theta^{r'}} = true$ and $TIME^{s'} < a.useBy^{\Theta^{r'}}$, i.e. the commit-event will only be executed after the publication of a positive reply and before the deadline expires;*
4. *If $a\checkmark \in EXC^r$ then there is no $r'' < r$ such that $a\text{✗} \in EXC^{r''}$, i.e. the commit-event will only be executed if the cancel-event was not executed;*
5. *If $a\text{✗} \in EXC^r$ then there is $r' < r$ such that $a\boxtimes \in PUB^{r'}$, $a.reply^{\Theta^{r'}} = true$ and $TIME^{s'} < a.useBy^{\Theta^{r'}}$, i.e. the cancel-event will only be executed after the publication of a positive reply and before the deadline expires;*
6. *If $a\text{✗} \in EXC^r$ then there is no $r'' < r$ such that $a\checkmark \in EXC^{r''}$, i.e. the cancel-event will only be executed if the commit-event was not executed;*
7. *If $a\boxtimes \in PUB^r$ and $a.reply^{\Theta^r} = true$, then there is no $r' = n_1 \longrightarrow n_2$ with $r < r'$ such that:*
   - $a\checkmark \in DSC^{r'}$
   - $TIME^{n_1} < a.useBy^{\Theta^r}$
   - *there is no transition $r < r'' < r'$ such that $a\text{✗} \in EXC^{r''}$*

i.e. the commit-event will not be discarded after a positive reply was executed unless the deadline has expired or the cancel-event has been executed;

8. *If* $a\boxtimes \in PUB^r$ *and* $a.reply^{\Theta^r} = true$*, then there is no* $r' = n_1 \longrightarrow n_2$ *with* $r < r'$ *such that:*

   - $a\boldsymbol{\times} \in DSC^{r'}$
   - $TIME^{n_1} < a.useBy^{\Theta^r}$
   - *there is no transition* $r < r'' < r'$ *such that* $a\checkmark \in EXC^{r''}$

   i.e. the cancel-event will not be discarded after a positive reply was executed unless the deadline has expired or the commit-event has been executed;

9. $a.pledge \in PLG^{s'}$ *if the following conditions hold:*

   - *there is a transition* $r' \in R$ *such that* $r' \leq s'$ *and* $a\boxtimes \in PUB^{r'}$ *and* $a.reply^{\Theta^{r'}} = true$
   - *there is no* $r'' < s'$ *with* $a\checkmark \in EXC^{r''}$ *or* $a\boldsymbol{\times} \in EXC^{r''}$
   - $TIME^{s'} < a.useBy^{\Theta^{r'}}$

   i.e. the pledge holds from the moment a positive reply is published until either the commit or the cancel are executed or the deadline expires;

10. *If* $a\boldsymbol{\Uparrow} \in PRC^r$*, then there is* $r' \in R$ *such that* $r' < r$ *and* $a\checkmark \in PRC^{r'}$*, i.e. the revoke-event will not be processed before the commit-event.*

A provider always replies to a request, but naturally 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) after and only after it has published a positive reply to the request and while the deadline does not expire — this is captured by clauses 3 to 8; the pledge that is associated with the positive reply will remain true until the deadline expires or the co-party commits or cancels — this is captured by clause 9.

Finally, every provider has a policy of not processing a revoke event before processing the associated commit event — this is captured by clause 10.

Clause 10 is necessary in order 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.

Chapter 4

The logic of service-oriented computing

In Chapter 3 we have characterized service-oriented computations. In particular, we have defined the notions of configuration and model of computation for a configuration. In this chapter, we discuss how the logic UCTL can be used to reason about such models and we give a logic characterization of service-oriented computations using UCTL. We start by presenting UCTL in Section 4.1. We proceed, in Section 4.2, by refining the syntax and semantics of UCTL with the primitives of service-oriented computation. Finally, in section 4.3, we use UCTL to axiomatize service-oriented computation.

Further ahead, in Chapter 5, we will see that the language that SRML provides for specifying service interfaces is based on UCTL. We will also see that there is a small difference between the logic that is presented in this chapter and the one that is used for specifications: here we define UCTL over the interactions associated with a configuration, while in specifications we use a logic that is defined over interaction names that are local to the specification of each of the nodes of a configuration. Nonetheless, the logic used in specifications (which is defined in Section 5.3) is equivalent to the one defined in this chapter in the sense that it uses the same set of operators (with the same semantics) and therefore is also referred to as "UCTL" throughout the thesis.

4.1 Background: the logic UCTL

In [10] UCTL was presented as an action/state-based temporal logic over Doubly Labelled Transition Systems. UCTL allows expressing properties that concern both the state of a computational system and the actions that it performs; this makes it easy to express properties that would be troublesome to write with pure action-based or state-based logics [10]. In this section, we review UCTL.

4.1.1 Doubly Labelled Transition Systems

UCTL is interpreted over *Doubly Labelled Transition Systems (L^2TS)*, which are transition systems in which both the states and the transitions between

states are labelled [28]. The L^{2}TSs 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 two-way interactions (see Section 3.3) is tipically published in the transition during which the associated request-event is executed, as an effect of that execution (this is discussed in Section 5.2).

Definition 4.1.1 (Doubly Labelled Transition System) *A* Doubly Labelled Transition System *(*L^{2}TS *for short) is a tuple*

$$\langle S, s_0, Act, R, AP, L \rangle$$

where:

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

The relation R defines the state transitions that can occur in the system and the actions that can be observed during those transitions. The state labelling function L defines which propositions, from a fixed set of atomic propositions AP, are true in each state.

A path from a state s in a L^{2}TS is a sequence of transitions that originates in s and represents one possible evolution when the system is in that state. The notion of path, path concatenation and path length are required for defining the semantics of UCTL and are defined as follows:

Definition 4.1.2 (Path) *Let $\langle S, s_0, Act, R, AP, L\rangle$ be an L^2TS and let $s \in S$:*

- *σ is a* path *from s if*
  - *$\sigma = s$ (the* empty path *from s) or*
  - *$\sigma = (s \xrightarrow{\alpha_1} s_1)(s_1 \xrightarrow{\alpha_2} s_2)\ldots(s_{i-1} \xrightarrow{\alpha_i} s_i)\ldots$, i.e. a possibly infinite sequence of transitions from s*

  We use $\sigma(i)$ to denote the state s_i and $\sigma(i-1,i)$ to denote the transition $s_{i-1} \xrightarrow{\alpha_i} s_i$.
- *The concatenation of paths σ_1 and σ_2, denoted by $\sigma_1\sigma_2$, is a partial operation, defined only if σ_1 is finite and its final state coincides with the first state of σ_2. Concatenation is associative and has identities: $\sigma_1(\sigma_2\sigma_3) = (\sigma_1\sigma_2)\sigma_3$ and if s_0 is the first state of σ and s_n is its final state, then $s_0\sigma = \sigma s_n = \sigma$.*
- *A path σ is said to be* maximal *if it is either an infinite sequence or it is a finite sequence whose final state has no successor states.*
- *The* length *of a path σ is denoted by $|\sigma|$ and defined as follows:*
  - *If $\sigma = s$, then $|\sigma| = 0$.*
  - *If σ is an infinite path, then $|\sigma| = \omega$.*
  - *If $\sigma = (s_0 \xrightarrow{\alpha_1} s_1)\cdots(s_n \xrightarrow{\alpha_{n+1}} s_{n+1})$ then $|\sigma| = n+1$.*

4.1.2 The syntax of UCTL

We proceed by defining the syntax of UCTL over a set of events *Act* and a set of atomic predicates *AP*. We use a to range over *Act* and p to range over *AP*. First, we define action formulas, which allow us to reason about transitions. Then we define the notion of UCTL formula for reasoning about L^2TSs (as a whole).

Definition 4.1.3 (Action formulas) *The syntax of* action formulas *is defined as follows:*

$$\chi ::= true \mid a \mid \tau \mid \neg\chi \mid \chi \wedge \chi$$

Definition 4.1.4 (UCTL formulas) *The syntax of UCTL formulas is defined as follows:*

$$\begin{aligned} \phi &::= true \mid p \mid \neg\phi \mid \phi \wedge \phi' \mid A\pi \mid E\pi \\ \pi &::= X_\chi \phi \mid \phi \,{}_\chi U\, \phi' \mid \phi \,{}_\chi U_{\chi'}\, \phi' \mid \phi_\chi W \phi' \mid \phi_\chi W'_\chi \phi' \end{aligned}$$

We refer to ϕ as state formulas *and to π as* path formulas.

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.

4.1.3 The semantics of UCTL

For the purposes of SRML, we define the semantics of action formulas in a slightly different manner than in [10]. Instead of interpreting action formulas over sets of actions, we interpret action formulas over transitions. We will see in Section 4.2.2 that this enables us to extend the syntax of action formulas with terms. The atomic formula *true* is satisfied by every transition. The atomic formula a (which represents an action) is satisfied by the transitions labelled by a set of actions that contains a, i.e. the transitions during which a occurs. The atomic formula τ is satisfied by transitions labelled by the empty set only and is used to represent unobservable actions.

Definition 4.1.5 (Satisfaction of action formulas) *The satisfaction relation for action formulas is defined as follows:*

- $s \xrightarrow{\alpha} s' \models true$
- $s \xrightarrow{\alpha} s' \models act$ *iff* $act \in \alpha$
- $s \xrightarrow{\alpha} s' \models \tau$ *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* $s \xrightarrow{\alpha} s' \models \chi'$

UCTL formulas can be state formulas, meaning they are interpreted over states, or path formulas, meaning they are interpreted over paths. The state formula *true* is satisfied by every state. The state formula p (where p is a predicate) is satisfied by the states labelled by p, i.e. the states in which p is true. The intuitive semantics of each of the operators used by UCTL is captured in Table 4.1. The formal semantics of UCTL formulas is given by the next definition.

Definition 4.1.6 (Satisfaction of UCTL formulas) *Let* $\langle S, s_0, Act, R, AP, L\rangle$ *be a* L^2 *TS. The satisfaction relation for UCTL formulas is defined as follows:*

- $s \models true$;
- $s \models p$ *iff* $p \in L(s)$;
- $s \models \neg\phi$ *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 *such that* $\sigma(0) = s$;
- $s \models E\pi$ *iff there exists a path* σ *with* $\sigma(0) = s$ *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* $0 \leq i$, $\sigma(i) \models \phi$ *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* $0 \leq i$, $\sigma(i) \models \phi$ *and* $\sigma(i, i+1) \models \chi$;

Other operators, like the diamond <> and box [ ] modalities of the Hennessy-Milner logic [43], can be seen as abbreviations of UCTL state formulas. In particular:

Definition 4.1.7 (Derived operators)

- $< \chi > \phi$ *stands for* $E[X_\chi \phi]$
- $[\chi]\phi$ *stands for* $\neg < \chi > \neg\phi$
- $EF\phi$ *stands for* $E[true_{true} U \phi]$
- $AF\phi$ *stands for* $A[true_{true} U \phi]$
- $EG\phi$ *stands for* $\neg AF \neg\phi$
- $AG\phi$ *stands for* $\neg EF \neg\phi$

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 Table 4.2.

$A\pi$	All paths from the state satisfy π.
$E\pi$	There exists a path from the state that satisfies π.
$X_\chi\phi$	The second (or next) state of the path is reached by a transition that satisfies χ and ϕ holds in that state.
$\phi_\chi U\phi'$	ϕ and χ hold in every state and transition of the path, respectively, until eventually there is state in which ϕ' holds.
$\phi_\chi U_{\chi'}\phi'$	ϕ and χ hold in every state and transition of the path, respectively, until eventually there is transition that satisfies χ' and leads to a state in which ϕ' holds.
$\phi_\chi W\phi'$	$\phi_\chi U\phi'$ holds in the path or ϕ and χ hold in every state and transition of the path, respectively.
$\phi_\chi W_{\chi'}\phi'$	$\phi_\chi U_{\chi'}\phi'$ holds in the path or ϕ and χ hold in every state and transition of the path, respectively.

Table 4.1: The intuitive semantics of the UCTL operators. The operators A and E are interpreted over a state, while the remaining operators are interpreted over a path.

$<\chi>\phi$	There is a transition from s that satisfies χ and ends in a state in which ϕ holds.
$[\chi]\phi$	Every transition from s satisfies χ and ends in a state in which ϕ holds.
$EF\phi$	There is a path that starts in s that leads to a state in which ϕ holds.
$AF\phi$	Every path that starts in s leads to a state in which ϕ holds.
$EG\phi$	There is a path that starts in s for which ϕ holds in every state.
$AG\phi$	ϕ holds in every state of every path that starts in s

Table 4.2: The intuitive semantics of the derived operators. Each of these operators is an abbreviation of a UCTL (state) formula interpreted over some state s.

4.2 Applying UCTL to service-oriented models

4.2.1 Service-Oriented Doubly Labelled Transition Systems

As discussed in Section 4.1, UCTL is interpreted over L^{2}TSs. The models of computation that we use for configurations are transition systems labelled with sets of events — what we call service-oriented transition systems (SO-TS) (see Def. 3.4.4). In order to use UCTL to reason about service-oriented models of computation we need to define a L^{2}TS-based abstraction of a SO-TS — what we call a *Service-Oriented L^{2}TS (SO-L^{2}TS)*.

A SO-TS and the associated SO-L^{2}TS have the same structure, i.e. they are formed by the same set of states and the states are connected in the same way. It is the labelling that is different for each of these two types of transition systems:

- The actions that label the SO-L^{2}TS correspond to the different stages of event propagation — publication, delivery, execution and discard (as discussed in Chapter 3). A transition of the SO-L^{2}TS is labelled by $e!$, $e¡$, $e?$ or $e¿$ if e is published, delivered, executed or discarded during that transition, respectively;

- Besides the pledges (see Section 3.3), each state of the SO-L^{2}TS also contains information about the actions that have happened before the system reached that state, i.e. the history of event propagation. Notice that while the history of actions can be derived by looking at the path that precedes the state, we need to make this information available directly from states in order to be able to reason about it — this is because UCTL does not contain operators to help reason about the past (see [46] for an example of a logic with past operators). Every state is also labelled by the function that interprets the parameters on that state.

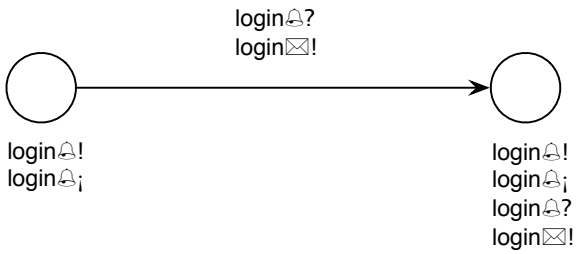


Figure 4.1: The SO-L^{2}TS transition that abstracts the transition shown in Figure 3.3. During this transition the event *login🔔* is executed and the event *login✉* is published; therefore the transition is labelled with *login🔔?* and *login✉!*. Since they occur during the transition, these two actions become part of the history of actions and therefore they also label the target state. Before the transition, the event *login🔔* had already been published and delivered (or else it could not be executed) and that is why the source state is labelled with *login🔔!* and *login🔔¡*.

As an example, in Figure 4.1 we show the (SO-L^{2}TS) transition that abstracts the (SO-TS) transition that was introduced in Figure 3.3.

Definition 4.2.1 (Service-oriented L^2TS) *The* Service-Oriented L^{2}TS *(*SO-L^{2}TS*) that abstracts a SO-TS $\langle S, \rightarrow, s_0, G\rangle$ is the tuple*

$$\langle S, s_0, Act, R, AP, L, TIME, \Pi\rangle$$

where:

- $Act = \{e! : e \in E\} \cup \{e¡ : e \in E\} \cup \{e? : e \in E\} \cup \{e¿ : e \in E\}$;
- $R \subseteq S \times 2^{Act} \times S$ *is such that:*
  - $s \rightarrow s'$ *iff* $(s, \alpha, s') \in R$ *for some* $\alpha \in Act^2$*, i.e. state interconnection is preserved;*
  - *For every* $(s, \alpha, s') \in R$*:*

    $$\alpha = \{e! : e \in PUB^{s\rightarrow s'}\} \cup \{e¡ : e \in ADLV^{s\rightarrow s'}\} \cup \{e? : e \in EXC^{s\rightarrow s'}\} \cup \{e¿ : e \in DSC^{s\rightarrow s'}\}$$

    i.e. transitions are labelled with the publication (!), delivery (¡), execution (?) and discard of events (¿);

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

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

  i.e. states are labelled by the history of event propagation and the pledges that are true in the state;
- *TIME assigns to each state* $s \in S$ *and transition* $s' \xrightarrow{\alpha} s$ *the instant* $TIME^s$;
- Π *assigns to each state* $s \in S$ *the parameter interpretation* Π^s.

It follows from Def. 4.1.1 that $\langle S, s_0, Act, R, AP, L\rangle$ *is a* L^2TS.

4.2.2 Extending UCTL

In order to be able to reason about the values of the parameters of events and time, we extend UCTL with terms. A UCTL term can be a constant, an operation, the value of some parameter or the time associated with a state.

Definition 4.2.2 (UCTL Terms) *The D-indexed family of sets* $TERM$ *is defined inductively as follows:*

- *If* $const \in F_d$ *then*

  $$const \in TERM_d$$

- *If* $f \in F_{<d_1,\ldots,d_n,d_{n+1}>}$ *and* $\vec{p} \in TERM_{<d_1,\ldots,d_n>}$ *then*

  $$f(\vec{p}) \in TERM_{d_{n+1}}$$

- *If* $p \in PP(e)_d$ *for some* $e \in E(a)$ *and* $a \in INT$, *then*

  $$a.p \in TERM_d$$

- $time \in TERM_{time}$

Definition 4.2.3 (Interpretation of terms) *The interpretation of a term $t \in TERM$ in state $s \in S$, written $[\![t]\!]_s$, is defined as follows:*

- $[\![const]\!]_s = const_{\mathcal{U}}$
- $[\![f(t_1, ..., t_n)]\!]_s = f_{\mathcal{U}}([\![t_1]\!]_s, ..., [\![t_n]\!]_s)$
- $[\![a.p]\!]_s = a.p^{\Pi(s)}$
- $[\![time]\!]_s = TIME(s)$

We extend action formulas with equations in order to be enable to write things like

$$EF < login\boxtimes ! \wedge login.Reply = true > true$$

which means that "there is a path on which a positive reply to a login request is eventually published".

Definition 4.2.4 (Extended action formulas) *The extended syntax of action formulas is defined as follows:*

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

As mentioned in Section 4.1, we interpret action formulas over transitions, instead of sets of actions (as in [10]). The reason for this is twofold: on the one hand we wish to be able to reason about the time at which an action took place (and in our model the time is associated with the states); and on the other hand we wish to reason about the parameters of the events that occur during the transition (i.e. the events that are published, delivered, executed or discarded) and the values of the parameters are fixed (after publication) in the states (see Def. 3.4.2). Because the parameters are fixed when the event is published (i.e. in the states that succeed the transition during which the event published), we need to interpret the terms of actions formulas over the target states of transitions.

Definition 4.2.5 (Satisfaction of extended action formulas) *The satisfaction relation for extended action formulas is defined as follows:*

- $s \xrightarrow{\alpha} s' \models t_1 = t_2$ *iff* $[\![t_1]\!]_{s'} = [\![t_2]\!]_{s'}$
- *All other formulas are interpreted as defined in Def. 4.1.5.*

We also introduce the syntactic category of state predicates, which consists of atomic predicates (i.e. the history of actions and pledges) and equations. This enables us to write things like

$$AG[login\text{✉}! \wedge login.Reply]\ time < login.useBy$$

which means that "the deadline for committing (or cancelling) after a positive reply to a login request is published will not expire immediately".

Definition 4.2.6 (State predicates) *The language SP of* state predicates *is defined as follows:*

$$SP ::= ap \mid t_1 = t_2$$

with $ap \in AP$ and $t_1, t_2 \in TERM_d$ for some $d \in D$

Definition 4.2.7 (Satisfaction of state predicates) *The satisfaction relation for state predicates is defined as follows, where $s \in S$:*

- $s \models ap$ *iff* $ap \in L(s)$
- $s \models t_1 = t_2$ *iff* $[\![t_1]\!]_s = [\![t_2]\!]_s$

We also need to extend the syntax of UCTL formulas by incorporating state predicates (instead of atomic predicates, as initially defined in Def. 4.1.4).

Definition 4.2.8 (UCTL formulas extended) *The extended syntax of UCTL formulas is defined as follows, where $sp \in SP$:*

$$\begin{aligned} \phi &::= true \mid sp \mid \phi \wedge \phi' \mid \neg\phi \mid A\pi \mid E\pi \\ \pi &::= X_\chi \phi \mid \phi\ {}_\chi U\ \phi' \mid \phi\ {}_\chi U_{\chi'}\ \phi' \end{aligned}$$

The satisfaction relation for the extended formulas is defined as follows:

- $s \models sp$ *as defined in Def. 4.2.7*
- *All other formulas are interpreted as defined in Def. 4.1.6.*

Given a model of computation for some configuration abstracted by the SO-L^{2}TS $\langle S, s_0, Act, R, AP, L\rangle$, we are able to use UCTL formulas, with syntax defined over Act and AP, to reason about that model.

4.3 The axioms of service-oriented computation

In Chapter 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. In particular, we have characterized the typical requester and provider conversation protocols that prevail in service-oriented computation. These properties can be expressed using formulas of UCTL. The advantages of having a set of formulas that characterize service-oriented computation are twofold: on the one hand it empowers us with a set of axioms over which we can reason; on the other hand it enables us to classify the nodes of a configuration by checking if they behave as requesters or providers, for each two-way interaction they are involved in.

The axioms of service-oriented computation are presented next as a set of theorems. In Appendix A we prove each of these theorems. Theorems 4.3.4 and 4.3.5 are of particular importance: we will see in Section 5.3, that these two theorems contain the set of formulas that give semantics to the interaction types $s\&r$ and $r\&s$ used in service interface specifications (like the one shown in Figure 1.2).

Let $t = \langle S, s_0, Act, R, AP, L, \Pi\rangle$ be the SO-L^{2}TS that abstracts some model of computation for a configuration $\langle N, WIRE, PLL, \Psi, 2WAY, 1WAY\rangle$.

Theorem 4.3.1 (Axioms of event propagation) *For every $e \in E$, s_0 satisfies the following formulas :*

- $A[true_{\{\neg e?\}}W_{\{e!\}}true]$
  (Events can only be delivered after they are published)

- $A[true_{\{\neg e?\wedge\neg e\not\subset\}}W_{\{e?\}}true]$
  (Events can only be processed after they are delivered)

- $AG\neg(e? \wedge e\not\subset)$
  (An event cannot be both executed and discarded)

Theorem 4.3.2 (Axioms of fairness) *For every $e \in E$, s_0 satisfies*

- $AG[e!]A[true_{\{true\}}U_{\{e?\}}true]$
  (After an event is published, it will eventually be delivered)

- $AG[e?]A[true_{\{true\}}U_{\{e?\vee e\not\subset\}}true]$
  (After an event is delivered, it will eventually be processed, i.e. it will be executed or discarded)

Theorem 4.3.3 (Axiom of session) *For every $e \in E$, s_0 satisfies*

- $AG[e!]A[true_{\{\neg e!\}}Wfalse]$
  (An event can only be published once)

- $AG[e? \vee e\not\subset]A[true_{\{\neg e?\wedge\neg e\not\subset\}}Wfalse]$
  (An event can only be processed once)

Theorem 4.3.4 (Axioms of Requesters) *A node $n \in N$ behaves as a requester in some interaction $a \in 2WAY_{\langle n,n'\rangle}$ iff s_0 satisfies the following formulas:*

1. $A[true_{\{\neg a\boxtimes?\}}W_{\{a\spadesuit!\}}true]$
   i.e. the reply-event will not be executed before the initiation-event is published;

2. $AG[a\spadesuit!]A[true_{\{\neg a\boxtimes\not\subset\}}Wfalse]$
   i.e. the reply-event will not be discarded after the initiation event was published;

3. $A[true_{\{\neg a\checkmark!\}}W_{\{a\boxtimes?\wedge a.Reply\}}true]$
   i.e. the commit-event will only be published after a positive reply was executed;

4. $AG[a\text{✗}!]A[true_{\{\neg a\checkmark!\}}Wfalse]$
   i.e. the commit-event will only be published if the cancel-event has not been published before;

5. $A[true_{\{\neg a\text{✗}!\}}W_{\{a\boxtimes?\wedge a.Reply\}}true]$
   i.e. the cancel-event will only be published after a positive reply was executed;

6. $AG[a\checkmark!]A[true_{\{\neg a\text{✗}!\}}Wfalse]$
   i.e. the cancel-event will only be published if the commit-event has not been published before;

7. $A[true_{\{\neg a\text{♔}!\}}W_{\{a\checkmark!\}}true]$
   i.e. the revoke-event will only be published after the commit-event was published.

Theorem 4.3.5 (Axioms of Providers) *A node $n \in N$ behaves as a provider in some interaction $a \in 2WAY_{\langle n,n'\rangle}$ iff s_0 satisfies the following formulas:*

1. $A[true_{\{\neg a\boxtimes!\}}W_{\{a\text{♞}?\}}true]$
   i.e. the reply-event will only be published after the initiation-event is executed;

2. $AG[a\text{♞}?]A[true_{\{true\}}U_{\{a\boxtimes!\}}true]$
   i.e. after the initiation-event is executed a reply-event will be published;

3. $A[true_{\{\neg a\checkmark?\}}W_{\{a\boxtimes!\wedge a.Reply\}}A[true_{\{\neg a\checkmark?\}}W_{\{a\checkmark?\wedge time<a.useBy\}}true]]$
   i.e. the commit-event will only be executed after the publication of a positive reply and before the deadline expires;

4. $AG[a\text{✗}?]A[true_{\{\neg a\checkmark?\}}Wfalse]$
   i.e. the commit-event will only be executed if the cancel-event was not executed before;

5. $A[true_{\{\neg(a\text{✗}?\}}W_{\{a\text{✉}!\wedge a.Reply\}}A[true_{\{\neg a\text{✗}?\}}W_{\{a\text{✗}?\wedge time<a.useBy\}}true]]$
   i.e. the cancel-event will only be executed after the publication of a positive reply and before the deadline expires;

6. $AG[a\text{✓}?]A[true_{\{\neg a\text{✗}?\}}W\,false]$
   i.e. the cancel-event will only be executed if the commit-event was not executed before;

7. $A[true_{\{\neg(a\text{✉}!\wedge a.Reply)\}}W_{\{a\text{✉}!\wedge a.Reply\}}\neg E[true_{\{\neg a\text{✗}?\}}U_{\{a\text{✓}\,¿\wedge time<a.useBy\}}]]$
   i.e. the commit-event will not be discarded after a positive reply was executed unless the deadline has expired or the cancel-event has been executed;

8. $A[true_{\{\neg(a\text{✉}!\wedge a.Reply)\}}W_{\{a\text{✉}!\wedge a.Reply\}}\neg E[true_{\{\neg a\text{✓}?\}}U_{\{a\text{✗}\,¿\wedge time<a.useBy\}}true]]$
   i.e. the cancel-event will not be discarded after a positive reply was executed unless the deadline has expired or the commit-event has been executed;

9. $AG[a\text{✉}!\wedge a.Reply](A[a.pledge_{\{true\}}W_{\{a\text{✓}?\vee a\text{✗}?\vee a.useBy\leq time\}}true])$
   i.e. the pledge must be true from the moment a positive reply is published until either the commit or the cancel are executed or the deadline expires;

10. $A[true_{\{\neg a\text{✞}?\wedge\neg a\text{✞}¿\}}W_{\{a\text{✓}?\vee a\text{✓}\,¿\}}true]$
    i.e the revoke-event will not processed before processing the commit-event.

Chapter 5

Specifying service-oriented systems

In Chapter 3 we defined a semantic domain for service-oriented computation. More precisely, we defined configurations as sets of interconnected computational entities that interact between each other by sending and processing typed events (like request, reply, commit, etc). We have also defined the notion of model of computation for a configuration as a state transition system, where the transitions are characterized by the sending, reception and processing of events by these computational entities.

In this Chapter we define a language for specifying service composition that is interpreted over the semantic domain defined in Chapter 3. As discussed in Section 1.4, the style of specification that we use is based on the SCA assembly model [12]. In SCA, new services are provided by interconnecting a set of components with a set of existing services and defining an interface to this system. The ultimate goal of this chapter is to define the notion of *service module* that was introduced in Section 1.4. In SRML, a service module is the (compositional) specification of a service through each of its parts: required services; internal components, which orchestrate the required services; and wires, which coordinate the interactions between the former. A service module defines a family of configurations and their models of computation, where each member represents a possible implementation of the intended service (when connected to a client). We proceed along the various sections of the chapter as follows:

5.1 defines how the interactions in which a party can be involved can be declared;

5.2 defines a state machine based language that is used for specifying the behaviour of the internal components of the service;

5.3 defines a temporal logic based language that is used for specifying service interfaces (i.e. the conversations of services with their clients);

5.4 defines a language for coordinating the interactions between pairs of parties, which is used for specifying wires;

5.5 formalizes the notion of service module and defines how this primitive is used for specifying service composition.

5.1 Interaction signatures

The specifications of every component and service interface have in common a declaration of the interactions in which the corresponding party (component or service) can be involved — what we call an *interaction signature* (see Def. 5.1.2). An interaction signature defines a set of local interaction names, together with their parameters, which denote the interactions in which a party can be involved. The names are local to the party being specified in order to allow specifications to be reused in different contexts (as for example different service modules) — the binding of the names that are used locally for each party is performed by the wire specifications. There is a type associated with each interaction name that allows us to derive which events can be published, received and processed by the party that is associated with the signature — this is essential for specifying when those events are in fact sent or processed by the party.

As an example, in Figure 5.1 we show the interaction signature of a *FlightAgent*, which is the type of one of the services required to compose *TravelBooking*. A *FlightAgent* can be involved in the interactions *lockFlight*, *payAck* and *payRefund*, which have types $r\&s$, rcv and snd, respectively. Each of these interactions has some parameters associated with the request-event (🔔). *lockFlight*, which is a two-way interaction, also has parameters associated with the reply-event (✉).

We will see further ahead that types $s\&r$ and $r\&s$ are used to denote two-way interactions from the point of view of the requester and provider, respectively. Types snd and rcv are used to denote one-way interactions from the point of the sender and the receiver, respectively. We will also see that if the signature is associated with a service interface then the type also defines the conversational protocol that the service follows. Throughout this thesis we restrict ourselves to using four interaction types, but we envision that further research can lead us to enrich this set of types in order to capture a wider variety of conversation protocols.

Definition 5.1.1 (Interaction Types) *Interactions can be of one of the following types* $TYPE = \{s\&r, r\&s, snd, rcv\}$.

```
INTERACTIONS
   r&s lockFlight
      🔔 from,to:airport,
         out,in:date,
         traveller:usrdata
      ✉ fconf:fcode
         amount:moneyvalue,
         beneficiary:accountn,
         payService:serviceId
   rcv payAck
      🔔 proof:pcode
         status:bool
   snd payRefund
      🔔 amount:moneyvalue
```

Figure 5.1: The interaction signature of a *FlightAgent*

Definition 5.1.2 (Interaction Signature) *An* interaction signature *is a pair*

$$\langle NAME, PARAM \rangle$$

where:

- *NAME is a TYPE-indexed family of sets of interaction names;*
- *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 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 sets of* ✉*-parameters,* ✓*-parameters,* ✗*-parameters and* ⇧*-parameters, respectively, such that* $Reply \in PARAM_{✉}(a)_{boolean}$ *and* $useBy \in PARAM_{✉}(a)_{time}$*;*
  - *For every* $a \in NAME$ *and* $P, P' \in PARAM_{🔔}(a) \cup PARAM_{✉}(a) \cup PARAM_{✓}(a) \cup PARAM_{✗}(a) \cup PARAM_{⇧}(a)$, P *and* P' *are disjoint.*

That is, an interaction signature defines a set of typed interaction names and a set of parameters for each of the events of the interaction. ✉-, ✓-, ✗-

and ✞-parameters (which are associated with the reply-, commit, cancel- and revoke events) are defined for two-way interaction names only (i.e. interaction names that are typed by either $s\&r$ or $r\&s$).

Throughout the remaining of Section 5.1 we consider a fixed interaction signature

$$s = \langle NAME, PARAM \rangle$$

over which all definitions will be given.

As already mentioned, the types that are associated with the interactions in a signature determine which events can be published by the party and which can be received and processed. That is, the types define the role of the party in interactions. For example, *FlightAgent*, which is involved in interaction *lockFlight* of type $r\&s$, can perform the action of publishing the reply-event of that interaction — denoted by *lockFlight*✉! — and the actions of receiving (i.e delivering), executing and discarding the other events — denoted by *lockFlight*🔔¡, *lockFlight*🔔?, *lockFlight*🔔¿!, *lockFlight*✓¡, and so on. The action of executing the reply-event of *lockFlight*, for example, is performed (under a different name) by the co-party of *FlightAgent* that plays the role of requester in *lockFlight*. The event names and action names that are associated with each interaction type are defined next.

Definition 5.1.3 (Event names) *The NAME-indexed families of sets* En^{PUB} *and* En^{RCV} *of names of events that can be published and received, respectively, is defined as follows:*

If $a \in NAME_{s\&r}$ *then* $En^{PUB}_a = \{a🔔, a✓, a✗, a✞\}$ *and* $En^{RCV}_a = \{a✉\}$*;*

If $a \in NAME_{r\&s}$ *then* $En^{PUB}_a = \{a✉\}$ *and* $En^{RCV}_a = \{a🔔, a✓, a✗, a✞\}$*;*

If $a \in NAME_{snd}$ *then* $En^{PUB}_a = \{a🔔\}$ *and* $En^{RCV}_a = \emptyset$*;*

If $a \in NAME_{rcv}$ *then* $En^{PUB}_a = \emptyset$ *and* $En^{RCV}_a = \{a🔔\}$*;*

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

Every parameter $p \in PARAM_{\#}(a)$ *where* $\# \in \{\text{🔔}, \text{✉}, \checkmark, \text{✗}, \text{🔑}\}$ *is said to be a parameter of event* $a\#$*.*

Definition 5.1.4 (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 interaction signature* s*.*

In Chapter 3 we mentioned that when a provider gives a positive reply, it makes a pledge to guarantee a set of properties. For each interaction with type $r\&s$ in an interaction signature (meaning that the party can publish the reply of that interaction), there is a pledge that can be specified.

Definition 5.1.5 (Pledge names) *The set* $PLNames$ *of pledge names associated with interaction signature* s *is* $\{a.pledge : a \in NAME_{r\&s}\}$*.*

Each interaction name in a signature denotes an interaction in which some node of a configuration is involved (which can have a different name in the specification of the peer node). An *interaction interpretation* over a configuration defines which interaction each name denotes. Also, an interaction interpretation defines the event, action and parameter that each event, action and parameter names denote, respectively. Since the two nodes involved in an interaction can observe the parameters of that interaction differently, an interaction interpretation defines for each parameter a function that models how that parameter is observed by the node associated with the signature.

Definition 5.1.6 (Interaction Interpretation) *An* Interaction Interpretation *II for s over a configuration* $\langle N, W, PLL, \Psi, 2WAY, 1WAY\rangle$ *is an injective function that:*

- *assigns an interaction in* $2WAY \cup 1WAY$ *to each name in NAME such that:*
  - *for every* $a \in NAME_{s\&r} \cup NAME_{r\&s}$, $II(a) \in 2WAY$, *i.e. interaction names with type s&r or r&s denote two-way interactions;*
  - *for every* $a \in NAME_{snd} \cup NAME_{rcv}$, $II(a) \in 1WAY$, *i.e. interaction names with type snd or rcv denote one-way interactions;*
- *assigns an event in E to each event name in En such that for every* $a \in NAME$ *and event name with form a# for some* $\# \in \{🔔,✉,✓,✗,🔑\}$*;*

  $$II(a\#) = II(a)\#$$

- *assigns an action to each action name in Act such that for every* $a \in NAME$ *and action name with form a#%, for some* $\# \in \{🔔,✉,✓,✗,🔑\}$ *and* $\% \in \{!, ¡, ?, ¿\}$

  $$II(a\#\%) = II(a)\#\%$$

- *assigns a pair* $\langle p', view\rangle$ *to each parameter name p in PARAM, where:*
  - p' *is a parameter in PP such that:*
    - *if* $p \in PARAM_{🔔}(a)_d$ *then* $p' \in PP(II(a🔔))_d$
    - *if* $p \in PARAM_{✉}(a)_d$ *then* $p' \in PP(II(a✉))_d$, $II(Reply) = Reply$ *and* $II(useBy) = useBy$
    - *if* $p \in PARAM_{✓}(a)_d$ *then* $p' \in PP(II(a✓))_d$
    - *if* $p \in PARAM_{✗}(a)_d$ *then* $p' \in PP(II(a✗))_d$
    - *if* $p \in PARAM_{🔑}(a)_d$ *then* $p' \in PP(II(a🔑))_d$

    i.e. p' *is the parameter denoted by p*
  - $view : d_{\mathcal{U}} \longrightarrow d_{\mathcal{U}}$, *where* $p \in PARAM_d$, *is such that if* $p = Reply$ *or* $p = useBy$ *then* $view = id$, *i.e. view is the function that defines how the parameter is observed.*

It results from definitions 5.1.4 and 5.1.6 that 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.

In order for an interaction signature to be associated with one computational node only, it is necessary that all interaction names in the signature denote interactions in which that node is involved and even more so that all action names associated with the signature denote actions that can be performed by that node. A *local interaction interpretation* is an interaction interpretation that satisfies this property.

Definition 5.1.7 (Local Interaction Interpretation) *An* Interaction Interpretation *II for s over a configuration* $\langle N, W, PLL, \Psi, 2WAY, 1WAY\rangle$ *is said to be* local *to a node* $n \in N$ *iff:*

- *For every name* $a \in NAME_{s\&r} \cup NAME_{snd}$, $II(a) \in INT_{\langle n,n'\rangle}$ *for some* $n' \in N$, *i.e. all interaction names with types s&r or snd denote interactions initiated by n;*

- *For every name* $a \in NAME_{r\&s} \cup NAME_{rcv}$, $II(a) \in INT_{\langle n',n\rangle}$ *for some* $n' \in N$, *i.e. all interaction names with types r&s or rcv denote interactions initiated by some other node.*

The definition says, that in order for an interpretation to be local to a node n it is necessary that each name with a type associated with the action of initiating the interaction (either $s\&r$ or *snd*) be indeed interpreted by an interaction initiated by n; and that the names associated with any of the remaining types be interpreted by interactions initiated by some other node.

5.2 Business Roles: specifying components

Components are computational units that (together with the wires) are used for orchestrating a set of external services in order to compose a new service. That is, the components act as mediators between a set of independent external services and the client of the service being provided. It is also possible that no external service is required, in which case the interconnection of components provides the service itself.

Service modules specify the components that are used in the provision of a 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 contains a declaration of a set of attributes that model the state of the component and a set of transition specifications that model how that state can change. A transition specification defines: (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 the components to be underspecified, thus avoiding premature decisions about how those components should be implemented. To illustrate we show part of the business role of a *BookingAgent* in Figure 5.2.

In this section, we present the language of business roles. We start by defining the notion of attribute declaration. We then proceed to define the language of business roles including its interpretation over SO-TSs.

5.2.1 Attribute declaration

An *attribute declaration* defines a set of variables that model the state of the component — what we call *attributes* of the component. For example, the specification of a *BookingAgent* contains the following attribute declaration:

```
BUSINESS ROLE BookingAgent is

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

ORCHESTRATION
   local
      s:[START, LOGGED, QUERIED, FLIGHT_OK, HOTEL_OK,
        CONFIRMED, END_PAYED, END_UNBOOKED,
        COMPENSATING, END_COMPENSATED],login:boolean,
        traveller:usrdata, travcard:paydata

   transition Request
      triggeredBy bookTrip🔔
      guardedBy s=LOGGED
      effects s'=QUERIED
             ∧ bookTrip🔔.out>today ⇒ bookFlight🔔
             ∧ bookFlight🔔.from=bookTrip🔔.from
             ∧ bookFlight🔔.to=bookTrip🔔.to
             ∧ bookFlight🔔.out=bookTrip🔔.out
             ∧ bookFlight🔔.in=bookTrip🔔.in
             ∧ bookFlight🔔.traveller=traveller
             ∧ bookTrip🔔.out≤today ⇒ bookTrip✉
             ∧ bookTrip✉.Reply=False

   transition TripCommit
      triggeredBy bookTrip✓
      guardedBy s=HOTEL_OK
      effects s'=CONFIRMED
             ∧ bookFlight✓ ∧ bookHotel✓ ∧ payment🔔
             ∧ payment🔔.amount=bookFlight✉.amount
             ∧ payment🔔.beneficiary=
                 bookFlight✉.beneficiary
             ∧ payment🔔.originator=traveller
             ∧ payment🔔.cardNo=travcard
```

Figure 5.2: Part of the specification of a *BookingAgent* using the language of business roles.

```
local
   s:[START, LOGGED, QUERIED, FLIGHT_OK, HOTEL_OK,
      CONFIRMED, END_PAYED, END_UNBOOKED, COMPENSATING,
      END_COMPENSATED]; login:Boolean;
      traveller:usrdata; travcard:paydata
```

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

Definition 5.2.1 (Attribute declaration) *An* attribute declaration *VAR is a D-indexed family of disjoint sets (where D is the set of datatypes).*

An *attribute interpretation* for a SO-TS models the internal changes of the component as it computes by assigning a value to each attribute in each state of the SO-TS.

Definition 5.2.2 (Attribute interpretation) *An* attribute interpretation *Δ for an attribute declaration VAR over a SO-TS $\langle S, \rightarrow, s_0, G\rangle$ assigns to every state $s \in S$ and every attribute $v \in VAR_d$ an element $v^{\Delta(s)} \in d_{\mathcal{U}}$ (the value of the attribute on that state).*

Throughout the remaining of Section 5.2 we consider:

- $sig = \langle NAME, PARAM\rangle$ to be an interaction signature where Act is the set of actions associated with sig;
- VAR to be an attribute declaration.

In order to interpret the language, we also consider:

- $\Xi = \langle N, W, PLL, \Psi, 2WAY, 1WAY\rangle$ to be a configuration;
- II to be an interaction interpretation for sig over $2WAY \cup 1WAY$ local to some node $n \in N$;
- $m = \langle S, \rightarrow, s_0, G\rangle$ to be a SO-TS for Ξ;

- Δ to be an attribute interpretation for VAR over m.

Next, we 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.

5.2.2 Sub-language of states

State terms denote the values of the parameters of events and attributes in states. We use $a.param$ where $param$ is a parameter of some event e of interaction a and not $e.param$. We do this because all parameters associated with events of the same interaction have different names and therefore $a.param$ unambiguously refers to a parameter of e and not of another event.

Definition 5.2.3 (State Terms) *The D-indexed family of sets $STERM$ of* state terms *(where D is the set of datatypes we have fixed) is defined as follows:*

- *If $c \in F_d$ then*

  $$c \in STERM_d$$

  for every $d \in D$

- *If $f \in F_{<d_1,...,d_n,d_{n+1}>}$ and $\vec{p} \in STERM_{<d_1,...,d_n>}$ then*

  $$f(\vec{p}) \in STERM_{d_{n+1}}$$

  for every $d_1, ..., d_n, d_{n+1} \in D$

- *If $a \in NAME$ and $param \in PARAM(a)_d$, then*

  $$a.param \in STERM_d$$

  for every $d \in D$

- $time \in STERM_{time}$

- *If* $v \in VAR_d$, *then*

  $$v \in STERM_d$$

  for every $d \in D$

In order to interpret the value of a parameter it is necessary to consider how that parameter is observed by the component — this is defined by the interaction interpretation through the function *view*.

Definition 5.2.4 (Interpretation of state terms) *The interpretation of a state term* $t \in STERM$ *in a state* $s \in S$, *written* $[\![t]\!]_s$, *is defined as follows, where* $II(param) = \langle param', view \rangle$*:*

- $[\![c]\!]_s = c_{\mathcal{U}}$
- $[\![f(t_1, ..., t_n)]\!]_s = f_{\mathcal{U}}([\![t_1]\!]_s, ..., [\![t_n]\!]_s)$
- $[\![a.param]\!]_s = view(II(a).param'^{\Pi^s})$
- $[\![time]\!]_s = TIME^s$
- $[\![v]\!]_s = v^{\Delta(s)}$

The sub-language of states is used for specifying the values of parameters, time and attributes in a state.

Definition 5.2.5 (Language of States) *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* $d \in D$.

Definition 5.2.6 (Satisfaction for the language of states) *The state satisfaction relation for the language of states LS is defined as follows, where* $s \in S$*:*

- $s \models true$

- $s \models t_1 = t_2$ *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* $s \models \phi'$

5.2.3 Sub-language of effects

Effect terms extend the syntax of state terms with the term v'. Unlike state terms, which are interpreted over states, effect terms are interpreted over transitions. In particular, the term v denotes the value of attribute v in the source state of the transition being considered, while the term v' denotes the value of v in the target state. Analogously, terms *time* and *time*$'$ denote the time in the source and target states, respectively.

Definition 5.2.7 (Effect Terms) *The D-indexed family of sets ETERM of* effect terms *is defined inductively as follows:*

- *If* $c \in F_d$ *then*

  $$c \in ETERM_d$$

  for every $d \in D$
- *If* $f \in F_{<d_1,...,d_n,d_{n+1}>}$ *and* $\vec{p} \in ETERM_{<d_1,...,d_n>}$ *then*

  $$f(\vec{p}) \in ETERM_{d_{n+1}}$$

  for every $d_1, ..., d_n, d_{n+1} \in D$
- *If* $a \in NAME$ *and* $param \in PARAM(a)_d$, *then*

  $$a.param \in ETERM_d$$

  for every $d \in D$
- $time, time' \in ETERM_{time}$
- *If* $v \in VAR_d$, *then*

$$v, v' \in ETERM_d$$

for every $d \in D$

Definition 5.2.8 (Interpretation of effect terms) *The interpretation of an effect term* $t \in ETERM$ *in a transition* $s_1 \rightarrow s_2$*, written* $[\![t]\!]_{s_1 \rightarrow s_2}$*, is defined as follows, where* $II(param) = \langle param', view \rangle$*:*

- $[\![c]\!]_{s_1 \rightarrow s_2} = c_{\mathcal{U}}$
- $[\![f(t_1, ..., t_n)]\!]_{s_1 \rightarrow s_2} = f_{\mathcal{U}}([\![t_1]\!]_{s_1 \rightarrow s_2}, ..., [\![t_n]\!]_{s_1 \rightarrow s_2})$
- $[\![a.param]\!]_{s_1 \rightarrow s_2} = view(II(a).param'^{\Pi^{s_2}})$
- $[\![v]\!]_{s_1 \rightarrow s_2} = v^{\Delta(s_1)}$
- $[\![v']\!]_{s_1 \rightarrow s_2} = v^{\Delta(s_2)}$
- $[\![time]\!]_{s_1 \rightarrow s_2} = TIME^{s_1}$
- $[\![time']\!]_{s_1 \rightarrow s_2} = TIME^{s_2}$

The sub-language of effects is used for specifying the effects of state transitions, namely how the internal state of the component changes (by equating the values of its attributes in the source and target states) and which events are published by the component.

Definition 5.2.9 (Language of Effects) *The* 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* $pub \in En^{PUB}$ *(i.e. pub is an event that can be published).*

Definition 5.2.10 (Satisfaction for the language of effects) *The satisfaction relation for the language of effects LE is defined for every transition* $r \in \rightarrow$ *as follows:*

- $r \models true$

- $r \models t_1 = t_2$ *iff* $[\![t_1]\!]_r = [\![t_2]\!]_r$
- $r \models pub$ *iff* $II(pub) \in PUB^r$
- $r \models \neg\phi$ *iff not* $r \models \phi$;
- $r \models \phi \wedge \phi'$ *iff* $r \models \phi$ *and* $r \models \phi'$;

5.2.4 Transition specifications

Using the sub-languages of states and effects we can define how state transitions are specified for components. For example, a component of type *BookingAgent* can be involved in the following transition:

```
transition TripCommit
  triggeredBy bookTrip✓
  guardedBy s=HOTEL_OK
  effects s'=CONFIRMED
         ∧ bookFlight✓ ∧ bookHotel✓ ∧ payment🔔
         ∧ payment🔔.amount=bookFlight✉.amount
         ∧ payment🔔.beneficiary=
             bookFlight✉.beneficiary
         ∧ payment🔔.originator=traveller
         ∧ payment🔔.cardNo=travcard
```

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 effects of the transition will take place: the events *bookFlight*✓, *bookHotel*✓ and *payment*🔔 will be published (with some constraints on their parameters).

Definition 5.2.11 (Transition specification) *A* transition specification *is a triple*

$$\langle trigger, guard, effects \rangle$$

where:

- $trigger \in En^{EXC}$ *or* $trigger \in LS$ *specifies the event or the state change that triggers the transition, respectively;*

- *guard* $\in LS$ *specifies the state in which the party must be for the transition to take place;*
- *effects* $\in LE$ *specifies the effects of the transition (i.e. how the state changes and which events are published).*

A SO-TS m satisfies a transition specification in which the trigger is an event e if (and only if) in every transition of m that starts in a state in which the guard is true and during which e is processed, e is executed (i.e. not discarded) and the effects are observed. In a similar way, m satisfies a transition specification in which the trigger is a state condition if (and only if) every transition of m 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) is followed only by transitions that satisfy the specified effects. Formally:

Definition 5.2.12 (Transition satisfaction) *The SO-TS m satisfies a transition specification $\langle trigger, guard, effects \rangle$ iff for every transition $r = s_1 \rightarrow s_2$ the following two properties hold:*

- *If* $trigger \in En^{EXC}$, $II(trigger) \in PRC^r$ *and* $s_1 \models guard$, *then*

  $$II(trigger) \in EXC^r \text{ and } r \models effects.$$

- *If* $trigger \in LS$, *not* $(s_1 \models trigger)$, $s_2 \models trigger$ *and* $s_2 \models guard$, *then for every* $s_2 \rightarrow s_3 \in R$

  $$s_2 \rightarrow s_3 \models effects.$$

5.2.5 Business Roles

As discussed, a business role is a specification of a component that declares the set of interactions in which that component can be involved, the set of attributes that characterize the internal state of that component and a set of transition specifications.

Definition 5.2.13 (Business role) *A* business role *is a triple*

$$\langle sig, VAR, ORCH \rangle$$

where:

- *sig is an interaction signature;*
- *VAR is an attribute declaration;*
- *ORCH is a set of transition specifications for sig and VAR.*

A SO-TS satisfies a business role if and only if it satisfies every transition specification in that business role.

Definition 5.2.14 (Satisfaction of business roles) *The SO-TS m is said to satisfy a business role* $\langle sig, VAR, ORCH \rangle$ *iff* $m \models t$ *for every* $t \in ORCH$.

5.3 Business Protocols: specifying service interfaces

Service modules specify which external services are required in order to provide a new service. As discussed in Section 1.4, each requires-interface in a service module is typed by a specification of the properties that the corresponding external service needs to satisfy regarding the way it interacts with its client. Also, the provides-interface of the module is typed by a specification of the properties that the module offers at its interface. In SRML, the specification of service interfaces (requires- or provides-) is given by *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 places some constraints on the manner in which it does engage in those interactions. Figure 5.3 shows the business protocol followed by a *FlightAgent*. A business protocol is abstract in the sense that it does not specify the internal state or workflow of the service (like it is done by business roles for

components), but only a temporal correlation between the actions that the service performs.

In this section, we present the language of business protocols. First, we adapt the logic UCTL to reason over interaction signatures — that is, while in Chapter 4 we have defined UCTL over the interactions of a configuration, in this chapter we define UCTL over local interaction names and their parameters. Then, we present a set of event correlation patterns that are persistent in SOC and that we use as templates for specifying service interfaces.

Throughout Section 5.3 we consider:

- $\Xi = \langle N, W, PLL, \Psi, \mathit{2WAY}, \mathit{1WAY} \rangle$ to be a configuration;
- $sig = \langle NAME, PARAM \rangle$ to be an interaction signature, where Act is the set of actions associated with sig;
- II to be an interaction interpretation for sig over $\mathit{2WAY} \cup \mathit{1WAY}$ local to some node $n \in N$;
- $m = \langle S, s_0, Act', R, L, AP, \Pi \rangle$ to be the SO-L^{2}TS that abstracts some model of computation for Ξ; $\mathit{II}[Act] \subseteq Act'$ by definition (i.e. Act' is a superset of the actions that can be performed by node n);

5.3.1 UCTL for specifications

In Chapter 4, we have defined UCTL over the interactions of a configuration. In order to keep business protocols local to the services that they specify we need to adapt UCTL so that we can reason about interactions using their local names, i.e. the names declared in the interaction signature of the service. More precisely, we want action formulas and state predicates to be defined over the names of the actions that can be performed by the service. Next, we redefine UCTL action formulas and state predicates over the actions and parameters associated with interaction signature sig. The structure of UCTL formulas that was defined in Section 4.2.2 is kept unchanged.

```
BUSINESS PROTOCOL FlightAgent is
  INTERACTIONS
    r&s lockFlight
      🔔 from,to:airport,
         out,in:date,
         traveller:usrdata
      ✉ fconf:fcode
         amount:moneyvalue,
         beneficiary:accountn,
         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✞?
    lockFlight✞? ensures payRefund🔔!
```

Figure 5.3: The business protocol followed by a *FlightAgent*. A *FlightAgent* can engage in three interactions (*lockFlight*, *payAck* and *payRefund*) and its behaviour obeys four constraints.

Definition 5.3.1 (Terms) *The D-indexed family of sets TERM is defined inductively as follows:*

- *If* $c \in F_d$ *then*

  $$c \in TERM_d$$

  for every $d \in D$

- *If* $f \in F_{<d_1,...,d_n,d_{n+1}>}$ *and* $\vec{p} \in TERM_{<d_1,...,d_n>}$ *then*

  $$f(\vec{p}) \in TERM_{d_{n+1}}$$

  for every $d_1, ..., d_n, d_{n+1} \in D$

- *If* $a \in NAME$ *and* $p \in PARAM(a)_d$, *then*

  $$a.p \in TERM_d$$

  for every $d \in D$

- $time \in TERM_{time}$

Definition 5.3.2 (Interpretation of terms) *The interpretation of a term* $t \in TERM$ *in state* $s \in S$, *written* $[\![t]\!]_s$, *is defined as follows, where* $II(p) = \langle p', view \rangle$*:*

- $[\![c]\!]_s = c_{\mathcal{U}}$
- $[\![f(t_1, ..., t_n)]\!]_s = f_{\mathcal{U}}([\![t_1]\!]_s, ..., [\![t_n]\!]_s)$
- $[\![a.p]\!]_s = view(II(a).p'^{\Pi(s)})$
- $[\![time]\!]_s = TIME(s)$

Definition 5.3.3 (Action formulas) *The language of* action formulas *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* $d \in D$.

Definition 5.3.4 (Satisfaction of action formulas) *The satisfaction relation for action formulas is defined 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* $II(act) \in \alpha$;
- $s \xrightarrow{\alpha} s' \models \tau$ *iff* $\alpha = \emptyset$
- $s \xrightarrow{\alpha} s' \models \neg\chi$ *iff not* $\alpha \models \chi$
- $s \xrightarrow{\alpha} s' \models \chi \wedge \chi'$ *iff* $\alpha \models \chi$ *and* $\alpha \models \chi'$

Action names can also be used as state predicates, in which case they have a different meaning: a state satisfies an action name if the action that the name denotes has happened in the past.

Definition 5.3.5 (State predicates) *The language SP of* state predicates *is defined as follows:*

$$SP ::= act \mid pledge \mid t_1 = t_2$$

with $act \in Act$, $pledge \in PLNames$ *and* $t_1, t_2 \in TERM_d$ *for some* $d \in D$

Definition 5.3.6 (Satisfaction of state predicates) *The satisfaction relation for state predicates is defined as follows, where* $s \in S$*:*

- $s \models act$ *iff* $II(act) \in L(s)$
- $s \models pledge$ *iff* $II(a).pledge \in L(s)$
- $s \models t_1 = t_2$ *iff* $[\![t_1]\!]_s = [\![t_2]\!]_s$

UCTL formulas and their semantics have been defined in Chapter 4 (Def. 4.2.8). We recall the syntax of UCTL formulas for reference, where p is a state predicate and χ is an action formula:

$$\begin{aligned} \phi &::= true \mid p \mid \phi \wedge \phi' \mid \neg\phi \mid A\pi \mid E\pi \\ \pi &::= X_\chi \phi \mid \phi \,{}_\chi U\, \phi' \mid \phi \,{}_\chi U_{\chi'}\, \phi' \end{aligned}$$

5.3.2 Patterns of service-oriented behaviour

In order to facilitate the specification of service interface behaviour we define the language of *behaviour constraints* in terms of a set of patterns that capture common requirements in the context of service-oriented interactions. The patterns capture the conditions under which events are executed or discarded (if processed) and the conditions under which events are published. Each of the patterns is defined as an abbreviation of UCTL.

Definition 5.3.7 (Behaviour constraints) *The set ABRV of* behaviour constraints *that can be specified for sig is defined as follows:*

$$ABRV ::= initiallyEnabled\ e? \mid s\ enables\ e? \mid s\ enables\ e?\ until\ w \mid s\ ensures\ e!$$

where $e?, e! \in Act$ *and* $s, w \in SP$.

Definition 5.3.8 (Interpretation of behaviour constraints)

- *"initiallyEnabled e?" stands for*

$$A[true_{\{\neg e¿\}} W_{\{e?\}} true]$$

  (The event e will never be discarded);

- *"s enables e?" stands for*

$$A[\neg s_{\{\neg e?\}} W(s \wedge \neg EF < e¿ > true)]$$

  (Once s (first) becomes true e cannot be discarded ever again and before s becomes true e cannot be executed);

- *"s enables e? until w" stands for*

$$\Big(\neg E[\neg w_{\{true\}} U(s \wedge E[\neg w_{\{true\}} U_{\{e¿\}} true])]\Big) \wedge$$

$$\Big(AG(w \Rightarrow \neg EF < e? > true)\Big) \wedge \Big(A[true_{\{\neg e?\}} W s]\Big)$$

initiallyEnabled e?	The event e will never be discarded.
s enables e?	Once s (first) becomes true e cannot be discarded ever again and before s becomes true e cannot be executed.
s enables e? until w	e can only be executed, and cannot be discarded, after s (first) becomes true but only while w has never been true. Once w (first) becomes true e cannot be executed anymore.
s ensures e!	After s (first) becomes true e will be published, but not before.

Table 5.1: The intuitive semantics of the event correlation patterns that can be used to specify service interface behaviour. e is an event and s and w are state predicates.

(e can only be executed, and cannot be discarded, after s (first) becomes true but only while w has never been true. Once w (first) becomes true e cannot be executed anymore.);

- "*s ensures e!*" *stands for*

$$A[\neg s_{\{\neg e!\}} W(s \wedge AF[e!]true)]$$

(After s (first) becomes true e will be published, but not before);

Table 5.1 summarizes the intuitive semantics of each of the event correlation patterns captured by the language of behaviour constraints — each of which can be used (in business protocols) to specify service interface behaviour. The set of patterns that we present is not intended to be complete; we envision that further research will lead us to extend (or adapt) this set of patterns.

5.3.3 Business Protocols

A business protocol is an interaction signature together with a set of behaviour constraints — expressed using the patterns defined in 5.3.7 and 5.3.8

and summarized in Table 5.1 — that correlate the actions associated with that signature.

Definition 5.3.9 (Business Protocol) *A* business protocol *is a pair*

$$\langle sig, BHV \rangle$$

where:

- *sig is an interaction signature;*
- *$BHV \subset ABRV$ is a set of behaviour constraints defined over sig.*

The fact that an interaction is typed with $s\&r$ or $r\&s$ in a business protocol means that the service (specified by that business protocol) engages in that interaction following the protocol of requesters or providers, respectively, which we have discussed in Section 3.5. For example, a service of type *FlightAgent* (shown in Figure 5.3) behaves as a provider in the interaction *lockFlight*, i.e. it always replies to a request, it waits for a commit or cancel, and so on. In order for a model of computation (abstracted by a SO-L^{2}TS) to satisfy a business protocol, that model must necessarily satisfy the set of UCTL formulas that characterize the behaviour of requesters or providers for each interaction declared in that business protocol with type $s\&r$ or $r\&s$, respectively. These are the formulas in Theorems 4.3.4 and 4.3.5.

A model of computation satisfies the specification given by a business protocol if and only if it satisfies the protocols associated with the types of each interaction declared in that business protocol and satisfies the event correlation specified by the behaviour constraints of that business protocol.

Definition 5.3.10 (Satisfaction of Business Protocols) *The SO-L^{2}TS m satisfies a business protocol $\langle \langle NAME, PARAM \rangle, BHV \rangle$ iff:*

- *For each $a \in NAME_{s\&r}$, m satisfies the set of formulas defined in 4.3.4, i.e. the formulas that characterize the requester protocol;*
- *For each $a \in NAME_{r\&s}$, m satisfies the set of formulas defined in 4.3.5, i.e. the formulas that characterize the provider protocol;*

- *For each* $x \in BHV$, $s_0 \models x$, *i.e.* s_0 *satisfies every behaviour constraint.*

At the end of this chapter, we discuss how business protocols are used to specify the interfaces of services required for the composition of a new service and also the interface that is provided by the new service.

5.4 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. In particular we define the notion of *interaction protocol*, which is the specification primitive for correlating pairs of interaction signatures.

Throughout the section we consider a fixed pair of interaction signatures $sig_1 = \langle NAME, PARAM \rangle$ and $sig_2 = \langle NAME', PARAM' \rangle$.

5.4.1 Syntax

The terms used for specifying wires, which we call *coordination terms*, have the same structure as the UCTL terms used in business protocols (defined in 5.3.1) — the difference is that coordination terms are defined over a pair of interaction signatures (not just one) and therefore can be used for correlating the parameters observed by two different parties.

Definition 5.4.1 (Coordination term) *The D-indexed family of sets $CTERM$ of* Coordination Terms *is defined as follows:*

- *If* $c \in F_d$ *then*

$$c \in CTERM_d$$

- *If* $f \in F_{<d_1,\ldots,d_n,d_{n+1}>}$ *and* $\vec{p} \in CTERM_{<d_1,\ldots,d_n>}$ *then*

  $f(\vec{p}) \in CTERM_{d_{n+1}}$

- *If* $a \in NAME$ *and* $param \in PARAM(a)_d$, *then*

  $a.param \subseteq CTERM_d$

  for every $d \in D$

- *If* $a \in NAME'$ *and* $param \in PARAM'(a)_d$, *then*

  $a.param \subseteq CTERM_d$

  for every $d \in D$

We use the operator $\equiv$ to specify that two interaction names are equivalent (in the sense that they refer to the same interaction). A *coordination* is a set of equations (on parameters) and equivalences between interaction names, such that each name cannot be equivalent to more than one other name. That is, a coordination defines a one-to-one relationship between the interaction names of the two signatures, but it does not necessarily pair every name. More precisely:

Definition 5.4.2 (Coordination) *A* Coordination *COORD is a set of elements of* ϕ, *where:*

- $\phi ::= t_1 = t_2 \mid a \equiv b$

with $t_1, t_2 \in CTERM_d$ *for some* $d \in D$ *and* $a \in NAME$ *and* $b \in NAME'$, *such that:*

- *For every* $a \in NAME$ *and* $b, c \in NAME'$ *if* $a \equiv b \in COORD$ *then* $a \equiv c \notin COORD$
- *For every* $a \in NAME'$ *and* $b, c \in NAME$ *if* $a \equiv b \in COORD$ *then* $a \equiv c \notin COORD$

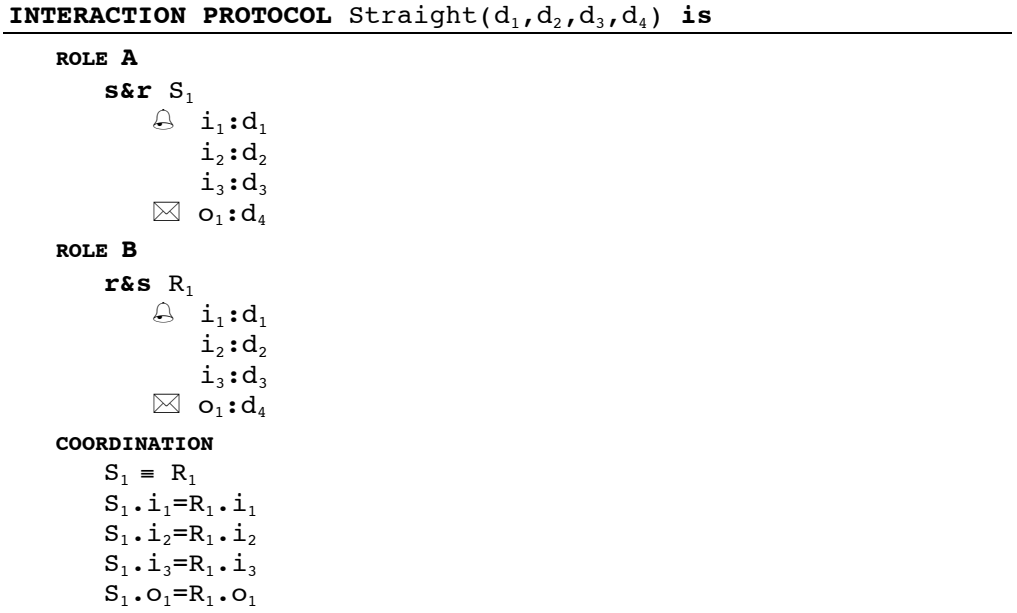


Figure 5.4: An interaction protocol that connects a s&r to a r&s interactions where the initiation-event (🔔) has three parameters and the reply-event (✉) has just one. According to this interaction protocol the two interactions are equivalent and their parameters are observed in the same way from the point of view of the two parties involved in the interaction. d_1, d_2, d_3 and d_4, which are the datypes of the parameters, are left undefined until the interaction protocol is applied.

An *interaction protocol* is a pair of interaction signatures together with a coordination that pairs the interaction names in those signatures and correlates the values of the parameters associated with those names. We call the two signatures in an interaction protocol *Role A* and *Role B*. Figure 5.4 shows an interaction protocol that connects a s&r with a r&s interactions, where the initiation event (🔔) has three parameters and the reply event (✉) has just one. According to this protocol the parameters are the same from the points of view of the two interactions. The datatypes of the parameters are left undefined until the interaction protocol is applied (i.e. $Straight(d_1, d_2, d_3, d_4)$ stands for a family of interaction protocols).

Definition 5.4.3 (Interaction Protocol) *An* Interaction Protocol *is a triple*

$$\langle sig_1, sig_2, coord \rangle$$

where coord ∈ COORD is a coordination for the interaction signatures sig_1 *and* sig_2*. We refer to* sig_1 *and* sig_2 *as the roles of the interaction protocol* — Role A *and* Role B*, respectively.*

Since we wish interaction protocols to be reusable, we define them independently of the names that are used for specifying the parties that they connect. Because of this, we require a mechanism for mapping the roles of the interaction protocol to the interaction signatures of the two parties that we wish to connect. We call this mechanism a *signature morphism* (see Def. 5.4.4). A signature morphism preserves the structure of interaction signatures: a signature morphism *map* from a signature s to a signature s' maps each interaction name of s to an interaction name of s' with the same type. Also, it maps each parameter of every interaction a to a parameter of $map(a)$ — choosing always a parameter that is associated with the same event, i.e. 🔔-parameters are mapped into 🔔-parameters and ✉-parameters are mapped into ✉-parameters. It is important to notice that not every interaction of the target signature needs to be in the mapping of a signature morphism. This property allows the coordination of only a part of the interactions of two parties. This is necessary because each party may interact with more than one co-party.

Definition 5.4.4 (Signature morphism) *A* signature morphism*, map, 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* $a \in NAME_t$ *with* $t \in TYPE$ *an interaction name* $a' \in NAME'_t$*;*
- *assigns to each parameter* $p \in PARAM_{\#}(a)_d$*, where* $\# \in \{🔔, ✉, ✓, ✗, ⇧\}$*, a parameter* $p' \in PARAM'_{\#}(a')_d$ *such that* $map(a) = a'$*;*

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 signature morphism. In that way, a connector defines which interactions are actually being correlated by

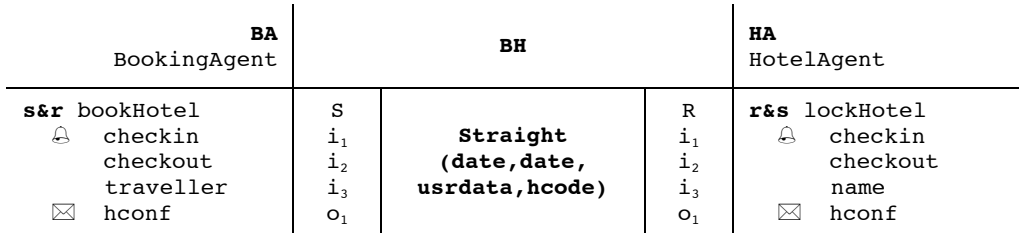


Figure 5.5: The connector that that binds the *BookingAgent* to the *HotelAgent* using a *Straight* interaction protocol. Variables S, $S.i_1$ $S.i_2$, $S.i_3$ and $S.o_1$ are associated with the interaction name *bookHotel* and parameters *checkin*, *checkout*, *traveller* and *hconf*, respectively, 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) are used to parameterize the interaction protocol.

the interaction protocol, thus allowing the interpretation of its coordination. Figure 5.5 shows the connector that binds the *BookingAgent* to the *HotelAgent* using the *Straight* interaction protocol shown in Figure 5.4. There is only one interaction between these two parties. Formally:

Definition 5.4.5 (Connector) *A* connector *for two interaction signatures partyA and partyB is a triple*

$$\langle map_1, ip, map_2 \rangle$$

where:

- $ip = \langle sig_1, sig_2, coord \rangle$ *is an interaction protocol;*
- map_1 *is a signature morphism from* sig_1 *to partyA;*
- map_2 *is a signature morphism from* sig_2 *to partyB.*

Let $partyA = \langle NAME, \_, \_ \rangle$ *and* $partyA = \langle NAME', \_, \_ \rangle$*; we say that an interaction name* $a \in NAME \cup NAME'$ *is coordinated by the connector iff* $a \in map_1 \cup map_2$ *(i.e. a is in the mapping defined by the connector).*

5.4.2 Semantics

In order to define the semantics of coordinations we consider a configuration and a model of computation, given by a SO-TS, for that configuration. We also consider two interaction signatures interpreted over the configuration and a connector between them, for which we will the define the meaning of the coordination. Let:

- $\Xi = \langle N, W, PLL, \Psi, \mathit{2WAY}, \mathit{1WAY} \rangle$ be a configuration;
- partyA and partyB be two interaction signatures;
- II_1 and II_2 be local interaction interpretations over $\mathit{2WAY} \cup \mathit{1WAY}$ for partyA and partyB, respectively. We will use II to denote $\mathit{II}_1 \cup \mathit{II}_2$;
- $\langle \mathit{map}_1, \langle s_1, s_2, \_ \rangle, \mathit{map}_2 \rangle$ be a connector for partyA and partyB. We will use map to denote $\mathit{map}_1 \cup \mathit{map}_2$;
- $m = \langle S, \rightarrow, s_0, G \rangle$ be the SO-TS that models the computation of Ξ;

In order to interpret a coordination term it is necessary to consider which interaction name in the signatures of the parties is each name in the coordination mapped to. It is also necessary to consider which interactions interpret which interaction names, which parameter values each interaction has in each state and how this values are observed locally by each of the two sides involved in the interaction.

Definition 5.4.6 (Interpretation of coordination terms) *The interpretation of a coordination term $t \in \mathit{CTERM}$ in a state $s \in S$, written $[\![t]\!]_s$, is defined as follows, where $\mathit{II}(\mathit{param}) = \langle \mathit{param}', \mathit{view} \rangle$:*

- $[\![c]\!]_s = c_{\mathcal{U}}$
- $[\![f(t_1, ..., t_n)]\!]_s = f_{\mathcal{U}}([\![t_1]\!]_s, ..., [\![t_n]\!]_s)$
- $[\![a.\mathit{param}]\!]_s = \mathit{view}((\mathit{II}(\mathit{map}(a)).\mathit{param}'^{\Pi^s})$

Two coordination terms are equal if and only if their denotation is the same throughout every state of the model of computation. That is, the correlation between parameters specified in the interaction protocol needs to be constant as the system computes. Two names a and b are equivalent, written $a \equiv b$, if and only if they are mapped to two interaction names that have the same interpretation — the function of the symbol $\equiv$ in wire specifications is to specify that two interaction names, that are being used independently in the specifications of two different parties, actually denote the same interaction. The notion of equivalence between interaction names depends only on how interactions are interpreted and is independent of which model of computation is assumed.

Definition 5.4.7 (Satisfaction of Coordinations) *The satisfaction relation for coordinations by the pair $\langle m, II \rangle$ (i.e. the model of computation and the interaction interpretation) is defined as follows:*

- $\langle m, II \rangle \models t_1 = t_2$ *iff* $[\![t_1]\!]_s = [\![t_2]\!]_s$ *for every* $s \in S$*;*
- $\langle m, II \rangle \models a \equiv b$ *iff* $II(map(a)) = II(map(b))$.

5.5 Service Modules: specifying service composition

Throughout the chapter we have defined how components, service interfaces and wires are specified. Next, we define how the composition of a service can be specified using a graph labelled by component, service interface and wire specifications — what we call a *service module*. We also define a notion of correctness for service modules. The idea is that every configuration and model of computation that satisfies the composition of a service module (i.e. it satisfies the specification of each part of the module) should also satisfy the (provides) interface that is advertised by the module.

5.5.1 Service Modules

In Section 1.4 we have introduced the notion of service module. In particular, in Figure 1.1 we show the structure of the service module *TravelBooking*. A service module defines a wired interconnection of components and required services. A module also defines which of these parties connects directly to the client of the service. We distinguish between wires that connect to the client — external wires — and wires that are internal to the service composition — internal wires. A service module provides a specification for each component, external-service and internal wire. In particular, each component is specified by a business role, each external service is specified by a business protocol and each wire is specified by an interaction protocol (established by a connector).

Every service module defines a provides-interface that advertises the set of interactions (and properties of these interactions) that are provided by the service — this set is a sub-set of the interactions specified locally for the parties that are wired to the client. It is important to notice that while the specification of each party correlates the actions that can be performed by that party, the specification given by the provides-interface is for the service as a whole and therefore establishes a correlation between the actions performed by the different parties that compose the service.

The behaviour of the client and external wires is not specified in a service module because the properties that are advertised by the service module, through the provides-interface, should be independent of the behaviour of the client and the wires that the client chooses [38], i.e. those properties should be satisfied for every possible client and external wires.

Definition 5.5.1 (Service module) *A* service module *is a tuple*

$$\langle N, W, C, client, spec, prov \rangle$$

where:

- $\langle N, W \rangle$ *is a simple graph (undirected, without self-loops or multiple edges), where* N *is a set of nodes (the parties that compose the service*

and the client of the service) and the symmetric relation $W \subseteq N \times N$ *is the set of edges (the wires that interconnect the parties and the client);*

- *We distinguish between different types of nodes:*
  - $client \in N$ *is the client of the service;*
  - $P = N \setminus \{client\}$ *are the nodes whose composition provides the service — called* parties *— and consist of:*
    - $C \subseteq P$ *the components; and*
    - $R = P \setminus C$ *the (external) required services;*
- *We distinguish between two types of wires:*
  - $WW = \{\langle n, n' \rangle \in W : n, n' \neq client\}$ *are the wires that do not connect to the client — what we call internal wires.*
  - $WE = W \setminus WW$ *are the wires that connect to the client — what we call external wires.*
- *spec assigns:*
  - *a business role to each component* $c \in C$*;*
  - *a business protocol to each required service* $r \in R$*;*
    We use $sign(p)$ *to refer to the interaction signature (indirectly) assigned by spec to a party* $p \in P$*;*
  - *a connector for* $sign(p)$ *and* $sign(p')$ *to each internal wire* $\langle p, p' \rangle \in WW$ *such that every interaction name coordinated by that connector is not coordinated by any other connector in* $spec[WW]$*;*
- $prov = \langle \langle NAME, PARAM \rangle, bhv \rangle$ *is a business protocol — the provides-interface of the module — such that:*
  - $NAME \subseteq \bigcup NAME^{sign(p)}$ *with* $p \in P$ *and* $\langle client, p \rangle \in W$*;*
  - $PARAM_{\#}(a) = PARAM_{\#}^{sign(p)}(a)$ *for each* $p \in P$*,* $a \in NAME \cap NAME^{sign(p)}$ *and* $\# \in \{🔔, ✉, ✓, ✗, 🔑\}$*;*

i.e. prov is a subset of the interactions of the parties that are connected to the client. If $\langle client, p\rangle \in W$ *and* $sign(p) = \langle NAME', BELL', ENV'\rangle$*, we use* $prov_p$ *to denote* $\langle NAME \cap NAME', BELL \cap BELL', ENV \cap ENV'\rangle$*, i.e. the interactions provided by the service through party p;*

We use $wire(a)$ *to denote a wire* $w \in W$ *such that:*

- $w \in WW$ *and a is in the coordination of* $spec(w)$*; or*
- $w = \langle client, p\rangle$*,* $prov_p = \langle NAME, \_, \_\rangle$ *and* $a \in NAME$

i.e. $wire(a)$ *denotes the wire through which interaction a takes place.*

In order to interpret a service module we need to consider the several properties of service-oriented systems that it specifies and recall how this properties are represented in the semantic domain we have defined. In particular, in every module specification:

- the graph defines which parties compose the service and how they are interconnected, i.e. it defines a set of configurations (those that satisfy the graph);
- the wire specifications define constraints on how the interaction names can be interpreted and a correlation between the parameters of interactions, i.e. wire interpretations define a set of interaction interpretations and set of models of computation for the configurations (those that satisfy the coordination done by the wires);
- the component, the requires-interface and the provides-interface specifications define constraints on how the configurations can compute — in particular, on how the value of state attributes (of components) evolve during computation. That is, they define a set of models of computation and a set of attribute interpretations (those that satisfy every component, requires-interface and provides-interface specification in the module).

For these reasons, the notion of satisfaction for service modules is given for interpretation structures composed by a configuration, an interaction interpretation for each party in the module, a model of computation (SO-TS) and an attribute interpretation for each component.

Definition 5.5.2 (Interpretation structure) *An interpretation structure for a service module* $m = \langle N, W, C, client, spec, prov \rangle$ *is a tuple*

$$\langle \Xi, II, t, \Delta \rangle$$

such that:

- $\Xi = \langle N', W', PLL, 2WAY, 1WAY \rangle$ *is a configuration;*
- t *is a SO-TS for* Ξ*;*
- II *is a* P*-indexed family of interaction interpretations over* $2WAY \cup 1WAY$ *such that* II_p*, with* $p \in P$*, interprets the signature in* $spec(p)$ *(i.e. it interprets the interactions of party* p*);*
- Δ *is a* C*-indexed family of attribute interpretations over* t*, such that* Δ_c *interprets the attribute declaration in* $spec(c)$ *(i.e. it interprets the attributes of component* c*).*

Let $prov = \langle \langle NAME, PARAM \rangle, bhv \rangle$*. We use* II^{prov} *to denote* $II|_{NAME \cup PARAM}$*, i.e. the interpretation of the interactions in the provides-interface;*

If an interpretation structure satisfies the configuration of parties defined by the service module and the individual specification of each party, we say that it satisfies the *composition* of the module. If such an interpretation structure also satisfies the provides-interface of the module we say that it satisfies the *provision* of the module. Next, we define these two notions of satisfaction for service modules.

We consider $m = \langle N, W, C, client, spec, prov \rangle$ to be a module specification and $\Xi = \langle N', W', PLL, 2WAY, 1WAY \rangle$ to be a configuration.

Definition 5.5.3 (Composition satisfaction of a service module) *An interpretation structure* $sem = \langle \Xi, \mathit{II}, t, \Delta \rangle$ *is said to satisfy the composition of* m*, written* $sem \overset{compos}{\models} m$*, iff:*

1. $N = N'$, $W = W'$ *and* $PLL = R \cup \{client\}$;
2. *For each party* $p \in P$, II_p *is local to* p;
3. *For each party* $p \in P$ *and interaction name* $a \in NAME^{sign(p)} \cap NAME^{sign(prov)}$, $\mathit{II}(a) \in INT_{\langle p,client \rangle} \cup INT_{\langle client,p \rangle}$;
4. $\langle t, \Delta \rangle$ *satisfies* $spec(p)$*, for each party* $p \in P$;
5. $\langle t, \mathit{II} \rangle$ *satisfies* $spec(w)$*, for each wire* $w \in WW$.

Condition *1* states that a service module defines a configuration (in the sense of Def. 3.2.1) where each of the components is modelled by a sequential computational node and the client and each of the external services are modelled by distributed computational nodes. That is, the client and the external services can each process more that one event during the same computation step (see 3.4.2). This is justified because external services, being services themselves, are typically implemented by a distributed set of interconnected computational units as well (see [38]), which for the purposes of this thesis we abstract with a distributed node. The client that connects to the module can also be a complex system (that may use this service module to compose yet another service) and therefore is also modelled by a distributed node. Condition *2* guarantees that each specification $spec(p)$ for party $p \in P$ is defined exclusively over actions that the node that models p can perform. Condition *3* states that each interaction name in the specification of a party p that is advertised in the provides-interface must denote an interaction between p and the client. Conditions *4* and *5* state that each component, requires-interface and wire specification must be individually satisfied.

Definition 5.5.4 (Provision satisfaction of a service module) *Let* $sem = \langle \Xi, \mathit{II}, t, \Delta \rangle$ *be an interpretation structure for* m *such that* $sem \overset{compos}{\models} m$ *and*

let t' be the L^2TS that abstracts t. sem is said to satisfy the provision of m, written $sem \overset{provis}{\models} m$, *iff* $\langle \Xi, II^{prov}, t' \rangle \models prov$.

An interpretation structure satisfies the provision of a service module if it satisfies the properties that the service advertises — through the provides-interface — regarding its interaction with the client.

5.5.2 Correctness of service modules

In principle, a service module should be such that the properties related to the composition (of components, external-services and wires) entail the properties advertized by the module, in which case we say that the module is *correct*.

Definition 5.5.5 (Module correctness) *The service module m is said to be correct iff, for every module interpretation sem of m*

$$\textit{if } sem \overset{compos}{\models} m \textit{ then } sem \overset{provis}{\models} m.$$

From a software engineering point of view, it is very important to know if a given service module is correct before implementing it. This is because in such case, any (correct) implementation of the composition defined in the module specification is guaranteed to provide a service that interacts with its client as intended and advertized. In the next Chapter, we discuss how the model-checker UMC can be used to support the development of correct service modules.

Chapter 6

Model-checking service-oriented systems

In this Chapter we propose a methodology for using the UML based model-checker UMC to support the development of service modules that are correct in the sense of Section 5.5.2. In Section 6.1, we review the model-checker UMC and, in Section 6.2, we present a methodology for model-checking the correctness of service modules with UMC.

6.1 UMC model-checker: a formal review

UMC is a model-checker for UML state machine based models. A UMC model is a set of objects, each of which is typed by a class whose behaviour is described by a UML state machine. UMC assigns to each model a L^2TS (in the sense of Def. 4.1.1) that represents all the possible evolutions of that system of objects. The properties of a model can be verified by checking which UCTL formulas the corresponding L^2TS satisfies.

In this section we make a formal review of the syntax of UMC models (using the same style of notation we have been using so far) that covers the features of UMC that are relevant for supporting our framework and give an overview of the semantics of these models.

6.1.1 Syntax of UMC models: UML state machines

UMC supports integers and boolean values (and arrays of integers and booleans) as primitive datatypes — these are used to type the attributes and the parameters of the signals that each class of objects can receive.

Definition 6.1.1 (UMC Types) $T = \{int, bool, obj, int[], bool[], obj[]\} \subset D$ *(where D is the set of datatypes we have fixed) are the (primitive) datatypes supported by UMC.*

A *class signature* defines the signals that can be invoked on the objects of that class (by other objects or by the objects themselves) and defines the state attributes of each of those objects. Every signal may have a set of parameters.

Definition 6.1.2 (UMC class signature) *A* UMC class signature *is a triple*

$$\langle SGN, PARAM, VAR \rangle$$

where:

- *SNG is the set of signals that the objects of the class can receive (which may trigger a state transition);*
- *$PARAM$ assigns to each signal a T-indexed set of parameters;*
- *VAR is a T-index family of sets of attributes of the objects of this class.*

A *UMC class* is a class signature together with a specification of how the objects of that class change state when they receive signals (i.e. when signals are invoked on the objects). This specification of behaviour consists of a declaration of a set of states in which the objects of the class can be and the set of transitions that can occur between these states, i.e. what is called a *UML state machine*. UML provides a syntactic notion of state that simplifies the specification of stateflows by allowing parallelism and containment relations between states — we refer to these syntactic states as *UML states* (or simply as states when there is no ambiguity) to distinguish them from the notion of computation state that takes into consideration the values of the attributes of the object.

Each UML state can be *composite*, meaning it contains a set of other states (referred to as substates), or *simple* in which case it does not contain any other state. Composite states can be *parallel*, meaning that their substates are all active simultaneously (i.e. an object enters all of the substates at the same time), or *sequential* in which case only one substate can be active at any given moment. The set of UML states in which the objects of some class can be is specified with a *UML state declaration*:

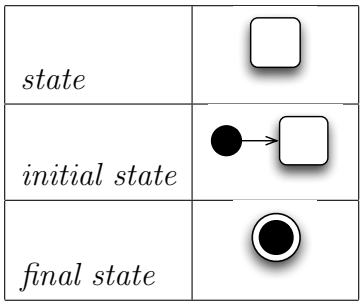
Definition 6.1.3 (UML state declaration) *A* UML state declaration *is a tuple*

$$\langle S, E, PLL, Initial, Final \rangle$$

where

- *S is a set of UML states;*
- *$\langle S, E\rangle$ is a tree, where S are the nodes and E are the edges, that represents the nesting of states; the leaf states (which have no nested states) are referred to as* simple states *— we use $LEAF$ to denote the leaf states;*
- *$PLL \subseteq (N \setminus LEAF)$ is the set of* composite parallel *states, whose children are referred to as* regions*; We refer to $N \setminus (LEAF \cup PLL)$ (the states that are neither simple nor parallel) as* composite sequential *states.*
- *$Initial, Final \subseteq LEAF$ are initial and final states, respectively.*

States are represented graphically as follows:



UML transitions are the syntactic constructs for specifying state transitions. A *UML transition* includes the same $\langle trigger, guard, effects\rangle$ structure as the transition specifications that we use in business roles (see Section 5.2), but the UML further uses states to restrict the ability of the transition to occur (and its effects). This is what in SRML corresponds to the use of a state attribute to model control flow, as is the case with attribute s of a *BookingAgent* (see Figure 5.2). That is, for a transition to happen (as the result of the trigger) it is not sufficient that the guard is true; the object must also be in some given source state. When it happens, a transition will take the object into a given target state. In UMC, the triggers, guards and effects of UML transitions are specified using a particular language, which for the sake of clarity will be defined after the (first class) notions of UML transition, UMC class and UMC model.

Definition 6.1.4 (UML transition) *A* UML transition *for a class signature* $\langle SGN, PARAM, VAR\rangle$ *and a UML state declaration* $\langle S, E, PLL, Initial, Final\rangle$ *is a tuple*

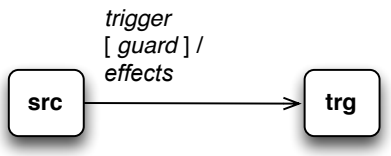
$$\langle src, trg, trigger, guard, effects\rangle$$

where:

- $src, trg \in S$ *are the source and target states of the transition, respectively;*
- *trigger is a UML trigger, i.e. the signal that triggers the transition;*
- $guard \in Cond$ *is the condition under which the transition can take place;*
- $effects \in Stm$ *are the actions performed if the transition takes place.*

The definition of UML trigger and the languages Cond (of UML conditions) and Stm (of UML statements) are introduced further ahead.

A UML transition is represented graphically as follows:



We can now formalize the notions of UML class and UML model that we have introduced informally:

Definition 6.1.5 (UML class) *A* UML class *is a tuple*

$$\langle SGN, PARAM, VAR, INI, S, T\rangle$$

where:

- $c = \langle SGN, PARAM, VAR\rangle$ *is a class declaration;*
- *INI is a function that assigns an initial value in* $t_{\mathcal{U}}$ *to each attribute in* VAR_t*;*

- *S is a UML state declaration;*
- *T is a set of UML transitions for c and S.*

Definition 6.1.6 (UMC model) *A* UMC model *is a pair*

$$\langle OBJ, class \rangle$$

where:

- *OBJ is a set of objects;*
- *$class$ assigns to each object in OBJ a UMC class.*

In summary, a *UMC model* consists of a set of objects each of which is typed by a class that specifies: the signals that the object can receive, the attributes that characterize the state of the object, the values of the attributes in the initial state of the object and the object's behaviour (given by a UML state machine).

The languages of transitions

UMC provides a sub-language of *conditions* and a sub-language of *statements*, which are used for specifying the guards and the effects of UML transitions, respectively. Both these languages are defined over the trigger of the transition, which is typically a parameterized signal. The parameters of the signal can be used as part of the guard or to specify the effects. More precisely:

Let $\langle OBJ, class \rangle$ be a UMC model and $cls = \langle \langle SGN, OP, PARAM, VAR \rangle, \_, T \rangle$ be the class of some object $obj \in OBJ$ of that model.

Definition 6.1.7 (UMC trigger) *A* UMC trigger *for a transition of cls can be:*

- *an empty trigger, written "–"; or*

- *s where $s \in SGN$, i.e. a signal.*

$LocalVar = PARAM(s)$ (or $LocalVar = \emptyset$ if the trigger is empty) is the set of variables that are associated with the trigger.

Let trg be a trigger for a transition of cls.

A UMC term can be a standard operation on integers (sum, subtraction, division or product), a parameter of the trigger or the value of some state attribute.

Definition 6.1.8 (UMC terms) *The T-indexed family of terms associated with trg is defined as follows:*

$$Term_{int} ::= intValue \mid Term_{int} + Term_{int} \mid Term_{int} - Term_{int} \mid Term_{int} * Term_{int} \mid Term_{int} \div Term_{int}$$

$$Term_t ::= LocalVar_t \mid Var_t$$

where $intValue \in F_{int}$, i.e. $intValue$ is an integer.

A UMC condition is a comparison between terms or the conjunction or negation of other conditions. Conditions are used — as guards — to specify in which (computation) states the object must be, and which properties the trigger must have, for a transition to take place.

Definition 6.1.9 (UMC conditions) *The language of* UMC conditions *for transition t is defined as follows:*

$$Cond ::= true \mid false \mid Term_t = Term_t \mid Term_{int} < Term_{int} \mid Cond \wedge Cond \mid \neg Cond$$

where $t \in T$ is some UMC type;

A UMC statement can be an assignment made to a state attribute (using the symbol ":=“), the sending of a signal to another object, a conditional “if then else” statement or a sequence of statements (separated by ";"). Statements are used for defining which changes are made to the state of the object and which signals are sent to other objects when a transition takes place.

Definition 6.1.10 (UMC statements) *The language of* UMC statements *for transition t is defined as follows:*

$$\begin{aligned} Stm ::= & \; Var_t := Term_t \mid obj.signal(p_1, \ldots, p_n) \\ & \mid if \; Cond \; then \; Stm \; else \; Stm \mid Stm \; ; Stm \end{aligned}$$

where $obj \in Obj$ is an object of the model.

6.1.2 Semantics of UMC models

Next, we give an overview of the semantics of UMC models, i.e. the function that assigns to every UMC model a L^2TS that represents the different ways in which that system of state machines can evolve.

Each of the objects that constitutes a UMC model has a queue that buffers the signals that the object receives. The state of that system of objects can be characterized by which signals are in the queues of the objects, by which UML states are active for each object and by the values of the attributes of the objects — this is the UMC equivalent to our notion of computation state (introduced in Chapter 3). More precisely:

Definition 6.1.11 (UMC computation state) *A* computation state *for a UMC model $\langle OBJ, class \rangle$ is a triple*

$$\langle INV, \Delta, Active \rangle$$

where:

- *INV is a OBJ-indexed family of sets of signals that are waiting to be processed in the queues of each object;*
- *Δ assigns to each attribute of type $t \in T$ of each object its value in $t_{\mathcal{U}}$;*
- *$Active$ is a OBJ-indexed family of sets of UML states which are active for each object, such that if $s \in Active_{obj}$ then s is in the state declaration of obj;*

The notion of "run-to-completion step", given by the UML specification [60], restricts the way in which individual UML state machines can evolve from one state to the other. UMC, which models the evolution of a system of several state machines, makes stronger assumptions than those defined by the UML standard — some of the assumptions made by UMC are that:

- only one object can evolve during each "run-to-completion step", i.e. UMC adopts an interleaving semantics for its models;
- the data structures that buffer the signals that an object receives are first-in-first-out queues;
- communication is instantaneous and reliable, i.e. signals are placed directly in the queue of the target object and no signal is lost in the process.

UMC associates with every model a L^2TS that represents all the possible evolutions of the system of objects that constitute the model. This L^2TS is generated on the fly — whenever some property needs to be checked against the model — in accordance with the notion of "run-to-completion step" (and in accordance with the other semantic assumptions that UMC makes). The states of this L^2TS are labelled by the computation states of the system and the transitions, which represent possible run-to-completion steps, are labelled by the actions performed during that step.

More precisely, if we take a model $\langle OBJ, class \rangle$ to be in a computational state s, the algorithm that generates the L^2TS transitions that originate from s consists of the following steps for every object $obj \in OBJ$:

1. Identify the enabled transitions — these are the transitions (specified by the class of the object) whose source state is active, whose trigger satisfies the signal at the front of the queue of obj (if there is one) and whose guard is true in s;
2. Select the transitions that can actually happen — these are the transitions with maximum priority according to a partial ordering of transitions based on the nesting of the source states [60]. More precisely,

a transition originating in some substate has a higher priority than a conflicting transition originating in a state that contains that substate;

3. Resolve conflicts: generate all the maximal subsets of transitions where there are no two transitions coming from intersecting states [60] — we call these the resolved subsets.

4. Serialize the transitions of each resolved subset, i.e. for each resolved subset generate all possible sequences of transitions — each of this sequences defines a possible 'run-to-completion step"/evolution and will generate its own branch in the L^2TS;

5. Compute the target computation state for each sequence of transitions by executing their effects sequentially — this includes removing the signal at the front the queue, updating the value of the attributes of the object and enqueuing each signal that was sent by the object in the buffer of the target object.

6.2 Using UMC to support service module development

The goal of a service provider, while developing a service module, is to guarantee that the orchestration of components and wires that will be implemented will provide the desired service once those components are connected (at run-time) to external services that satisfy the required behaviour. The main challenge involved in supporting this development process with the UMC model-checker results from the fact that in our framework the required services are specified with patterns of interface behaviour — those which we have characterized logically in Chapter 5 — and not with state machines. In order to model-check the orchestration performed by the components and wires of a module it is necessary to define state machines that satisfy the behaviour of the required services and interconnect them with the components. The properties that are expected of the resulting service can then be checked over that system of state machines.

Our contribution towards the process of model-checking service modules with UMC consists mainly of defining a catalogue of stateflow patterns that can be composed for encoding the behaviour of a service interface with a UMC state machine. Similarly, we define a catalogue of stateflow patterns through which the coordination performed by the wires can be encoded with UMC state machines. In this section, we present a methodology for using these patterns to encode a module specification with a system of UMC state machines.

6.2.1 Assumptions and limitations

The semantic domains of SRML (introduced in Chapter 3) and UMC/UML state machines are not the same; the differences between these two domains reflect the different levels of abstraction that are used to characterize service-oriented computation at the business level and to characterize a system of generic state-machines, respectively. The methodology that we propose for encoding (and model-checking) service modules with UMC/UML state machines is limited by the nature of these differences. The following are short discussions on some of the assumptions and limitations (or what at the surface appear to be limitations) of our methodology.

Real time Since UMC does not deal with real time, we restrict ourselves to model-checking a subset of SRML that does not deal with real time properties. More precisely, we assume that: (1) the state variable *TIME* is not part of the specification languages and therefore service modules do not contain properties about the time; and (2) there is no dealine associated with two-way interactions (see Chapter 3); the provider in a two-way interaction will remain ready to execute the commit- or the cancel-event indefinitely after publishing a positive reply.

Parameterization UMC forces us to define the parameter values with which each signal can be sent by each state machine. Because of this, a UMC encoding of a module specification requires a concrete parameterization

of each event that can be published — in this sense, a UMC encoding is a model of a particular session that is characterized by the parameter values of each event that can be published during that session. The properties that a UMC encoding satisfies depend on the values that are chosen to parameterize the events. This means that in order for the developer to be able to draw conclusions about the correctness of the module, it is essential that he/she parameterizes the events wisely.

Components SRML supports a style of engineering in which specifications are refined at different stages of the development process, such that in a final stage an implementation for the service is reached. The language of business roles (introduced in Chapter 5), which is used for specifying the components of a module — at an earlier stage of the development process —, allows defining the family of state machines out of which a member will be chosen — at a later stage of development — for implementation. The model-checking techniques we propose in this Chapter are aimed at supporting that later stage of development and therefore assume that the orchestration performed by each component of a service module can be and is already refined with a UML state machine.

External service distribution We assume that external services are not distributed. By encoding each requires-interface with a single UMC state machine we are adopting an implementation where only one event can be processed by the external service during each computation step (this is because each state machine has one and only one queue) — although typically a service results from the composition of several distributed components. From a practical point of view this is not a big limitation since the business logic of a service module typically does not require external services to execute two events simultaneously, but only that they do execute them.

Module distribution We assume that the composition of components, wires and external services is not distributed, i.e. only one of these parties can

evolve in each computation step. This limitation results from the interleaving semantics that UMC adopts for UML state machines. Again, from a practical point of view this is a small limitation because the properties offered by a service module typically do not depend on if two events are executed or published simultaneously, only on if they are indeed executed or published; nor do those properties depend on if the state of two different parties is updated simultaneously (in fact the latter cannot be observed with the logic we have defined for reasoning about services).

6.2.2 A methodology for encoding service composition with UMC/UML state machines

A UMC-encoding of a service module is a set of communicating state machines, each of which models a part of the system that is specified by that module. In particular, in the UMC encoding that we propose:

- there is a state machine for each component that models the orchestration performed by that component — we assume these are given a priori;
- there is a state machine for each requires-interface that models the possible interface behaviours of a service that satisfies the business protocol of that requires-interface;
- there is a state machine that models all the possible interface behaviours of a client of the service (in particular, it models all the requests that the client can make to the service);
- there is a state machine for each equivalence between interaction names (specified by the connectors of the wires) whose behaviour guarantees that events are correctly delivered. That is, they model the internal wires;
- there is a state machine for each interaction advertised in the provides-interface of the module; these set of state machines guarantee that

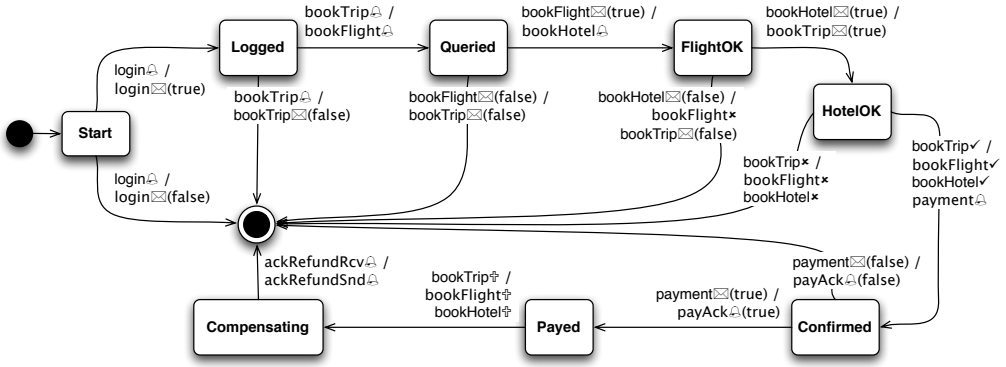


Figure 6.1: The UML state machine that models the orchestration performed by a *BookingAgent*. Parameters that do not affect the workflow are not shown.

events sent by the client to the service and vice versa are correctly delivered. That is, they model the wires that connect to the client.

Next, we discuss the methodology used to build each of these kinds of state-machine. Afterwards, we formalize the methodology by defining the notion of *UMC encoding of a module specification*.

Components

As discussed at the beginning of the section, we assume that the behaviour of each component is already refined as a UML state machine. For example, in order to model-check the running example *TravelBooking*, we take the state machine shown in Figure 6.1 as a model of the orchestration performed by the *BookingAgent* component *BA*.

Encoding requires-interfaces

As discussed in Chapter 5, the specification of a requires-interface consists of a typed declaration of the interactions that the required service can engage in and a set of behaviour constraints that correlate the events of those interactions. Our strategy for encoding a requires-interface as a state machine consists of including in the state machine a region (i.e. a composite parallel state) for each of the interactions in which the service can engage – which we

e_ensured	e needs to be published
e_enabled	e should be executed if processed
e_published	e has been published
e_executed	e has been executed

Table 6.1: The intuitive meaning of the boolean attributes associated with an event e — each of these is an attribute of the state machine that encodes the requires-interface in which the interaction associated with e is declared.

call *interaction-regions* – and a region for each of the behaviour constraints – which we call *constraint-regions* – except for the constraints defined with the pattern "*initiallyEnabled e?*" (that define that an attribute *e_enabled* is initialized to *true*).

The role of each of the interaction-regions is to guarantee that the conversational protocol that is associated with the type of the interaction is respected. Events of a given interaction are published and executed exclusively by the interaction-region that models that interaction. The role of the constraint-regions is to flag, through the use of special state attributes, when events should be published and when events become enabled for execution – the evolution of the interaction-regions, and thus the actual publication and execution of events, is guarded by the value of those flags. That is, constraint-regions cooperate with interaction-regions to guarantee the correlation of events expressed by the behaviour constraints.

In particular, for each event e that can be published by the external service there is an attribute *e_ensured* in the state machine — which can be used by constraint-regions to flag that e needs to be published — and for each event e that can be executed there is an attribute *e_enabled* — which can be used by constraint-regions to flag that e should be executed if processed. For each event e there is also an attribute *e_published* or *e_executed*, which is updated by the interaction-regions (when e is published or executed, respectively) and monitored by the constraint regions. Table 6.1 summarizes the meaning of the attributes associated with an event e.

According to this methodology, every interaction of the same type defines a region with the same pattern of workflow, where only the names of the

events are different; also, every behaviour constraint that follows the same correlation pattern defines a region with the same pattern of workflow, where only the conditions that guard the transitions are different. In Figure 6.2 we show the encoding of a *FlightAgent*. For example, the constraint *lockFlight*↑*? ensures payRefund*♠*!* is enforced by the state machine as follows: 1) when region A executes *lockFlight*↑ it sets the attribute *lockFlight_executed* to true; 2) this attribute is monitored by region Z (which encodes the constraint) which responds by setting the value of attribute *payRefund*♠*_ensured* to true; 3) as a consequence region C publishes the event *payRefund*♠.

The encoding that we propose for requires-interfaces is compositional in the sense that each region in the state machine is defined exclusively by the properties of the interaction or constraint that it encodes, regardless of the other interactions or constraints in the business protocol that specifies the requires-interface.

Encoding a generic client

A generic client is one that interacts with the service in every possible way that the service allows, i.e. it engages nondeterministically in the interactions declared in the provides-interface of the service. We encode such a client by creating a state machine with a single state and a loop transition for each event that the client can publish when interacting with the service (as specified by the provides-interface of the service module). Such a state machine can publish each event of each interaction at any given time.

Since the interactions that the client can engage in are not specified locally to the client, we use the names declared in the provides-interface of the module to encode those interactions — these are the same names that are used in the encoding of the co-parties of the client.

Encoding wires

In Chapter 3 we have seen that interactions are peer-to-peer, i.e. they take place between a pair of parties. Then, in Chapter 5 we have seen that the interactions each party can engage in are specified locally for the parties using

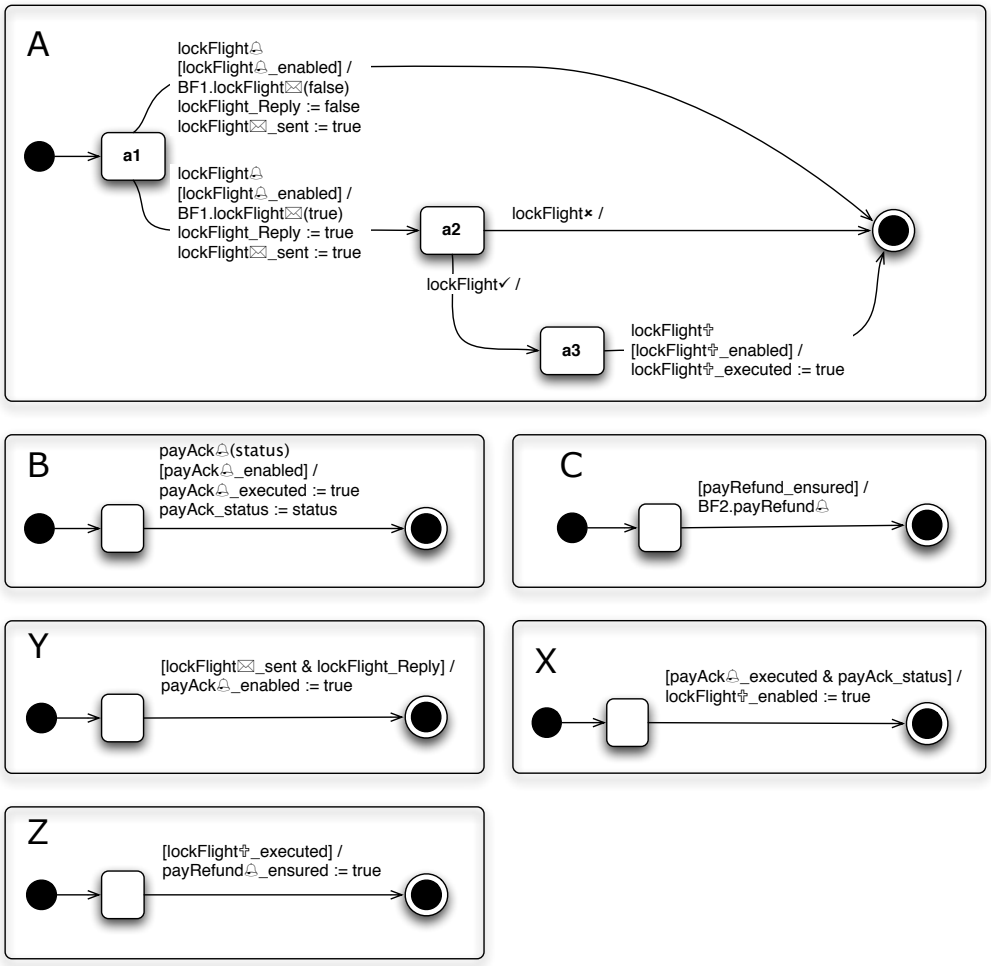


Figure 6.2: The UML state machine that encodes a requires-interface of type *FlightAgent*; the attribute initialization is not shown. A *FlightAgent* is involved in the three interactions *lockFlight*, *payAck* and *payRefund* that are encoded by interaction-regions A, B and C, respectively; these three interactions are correlated by four behaviour constraints that originate the three constraint-regions Y, X, and Z. The constraint "*initiallyEnabled lockFlight?*" does not originate a region in the state machine; instead it determines that the attribute *lockFlight_enabled* is initially set to *true*.

local names — in a module specification there are two different names that denote the same interaction, one name for each of the two parties involved in the interaction. The role of wire specifications is to pair the interaction names declared locally to the parties, thus defining which interactions take place between each parties.

In the encoding of module specifications we propose, each wire is encoded by a set of independent state machines: one for each interaction that occurs through that wire, i.e. one for each equivalence between interaction names that is in the specification of that wire. Each of these state machines is responsible for coordinating the events of the interaction it is associated with. The idea is that a party publishes an event of some interaction by signaling it in the state machine that coordinates that interaction; this state machine in turn forwards the event by signaling it in the co-party, using not the same, but the equivalent name (which is the name that the co-party recognizes) — in the particular case of the wires that connect to the client, the same names need to be used on both ends of the wire.

In the *TravelBooking* module two-way interactions are coordinated by straight interaction protocols that bind the names and parameters of $s\&r$ and $r\&s$ interaction declarations directly (i.e. events and parameter values are the same from the point of view of the two parties connected). Figure 6.3 shows the state machine that encodes this connector for the single interaction that takes place between *BA* and *HA* — there is only one persistent state in which the machine waits to receive events and forward them with the same parameter values.

This encoding, with multiple state machines instead of just one, together with the interleaving semantics of UMC, guarantees that all possible orders of delivery between events of different interactions are modelled. Notice, this would not be so if each wire was encoded by a single state machine because state machines have a first-in/first-out buffering policy that would enforce events to be delivered in the order in which they were published.

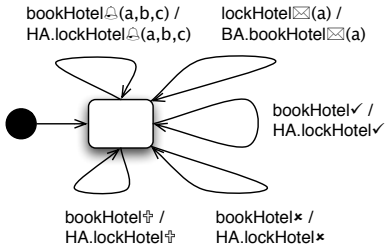


Figure 6.3: The UML encoding of the connector that coordinates the single (two-way) interaction between *BA* and *HA* which is named *bookHotel* and *lockHotel* from the point of each party, respectively.

Encoding UCTL predicates with UMC boolean conditions

Before we define how the composition of a module specifications is encoded with a UMC model, we need to define how UCTL state predicates — which are used for specifying constraints in business protocols (see Chapter 5) — are translated into UMC state conditions that can be used as guards in transitions. The goal of this translation is to be able to create state machines (for encoding requires-interfaces) that monitor when some state predicate becomes true. It is important to notice that it is only possible to translate those state predicates built with the datatypes (and operation over these datatypes) that are supported by UMC.

First we define how to encode UCTL terms. Every parameter name is translated into a variable with the same name, which (as we will see further ahead) stores the value of that parameter.

Definition 6.2.1 (Encoding of terms) *The partial function* $\omega : TERM \longrightarrow EXP^{UMC}$ *that translates UCTL terms into UMC expressions is defined as follows:*

- $\omega(const) = const$ *iff* $const \in F_{int}$*, i.e. const is an integer number;*
- $\omega(f(t_1, t_2)) = t_1\ f\ t_2$ *iff* $t_1, t_2 \in TERM_{int}$ *and* $f \in \{+, -, *, \div\}$*, i.e. binary operations over integers are translated into infix notation;*
- $\omega(a.param) = a\_param$*;*

The state predicates $e!$ and $e?$ are translated into the variables *e_published* and *e_executed*, respectively — these two variables are used to flag that event e has been published or executed, i.e. these variables model the history of events.

Definition 6.2.2 (Encoding of state predicates) *The function* $\Omega : SP \longrightarrow SF^{UMC}$ *that translates state predicates into UMC boolean conditions is defined as follows:*

- $\Omega(e!) = e\_published$
- $\Omega(e?) = e\_executed$
- $\Omega(t_1 = t_2) = \omega(t_1) = \omega(t_2)$

Encoding service composition with UMC

The following (lengthy) definition formalizes the methodology we have been discussing throughout the section. That is, it formalizes how the composition of components, wires and external services that is specified by a service module is encoded with a UMC model according to our methodology. In particular, it defines the UML patterns that encode: (1) the conversational protocols that are associated with each interaction type (see Section 5.3); (2) the event correlation patterns (see Section 5.3); and (3) the coordination of interactions performed by wires (see Section 5.4).

Definition 6.2.3 (Encoding of service composition) *An encoding of the composition specified by a service module*

$$\langle N, W, C, client, spec, prov \rangle$$

is a UMC model

$$\langle OBJ, CLASS \rangle$$

where OBJ is the minimum set such that:

- *For each participant (client, components and requires-interfaces)* $n \in N$*, there is an object* $o^n \in OBJ$ *that models it;*

- *For every equivalence $x \equiv y \in coord$ (which pairs interaction names), where $spec(w) = \langle map_1, \langle \_, \_, coord \rangle, map_2 \rangle$ for some wire $w \in WW$, there is an object $o^{x \equiv y} \in OBJ$, which is responsible for coordinating the events of the interaction defined by the equivalence; We also use $o^{map(x)}$ or $o^{map(x)}$ (where $map(x)$ and $map(y)$ are the actual interaction names being bound) to denote $o^{x \equiv y}$;*

- *For every interaction provided by the service $a \in NAME$, where $prov = \langle \langle NAME, \_, \_ \rangle, \_ \rangle$, there is an object $o^a \in OBJ$, which is responsible for coordinating the events of a (between the service and the client);*

The class $CLASS(o)$ of each object $o \in OBJ$ satisfies the following properties:

Components *Let $c \in C$ be a component, with $spec(c) = \langle \langle NAME, PARAM, VAR, \rangle, ORCH \rangle$. $CLASS(o^c) = \langle SGN, P, VAR, INI, SD, T \rangle$ is such that:*

- *$SGN = EN^{RCV}$, i.e. there is a signal for each event that can be received by the component;*
- *$P(e) = \{p \in PARAM : p \text{ is a parameter of } e\}$ i.e. signals have the same parameters as the corresponding events;*

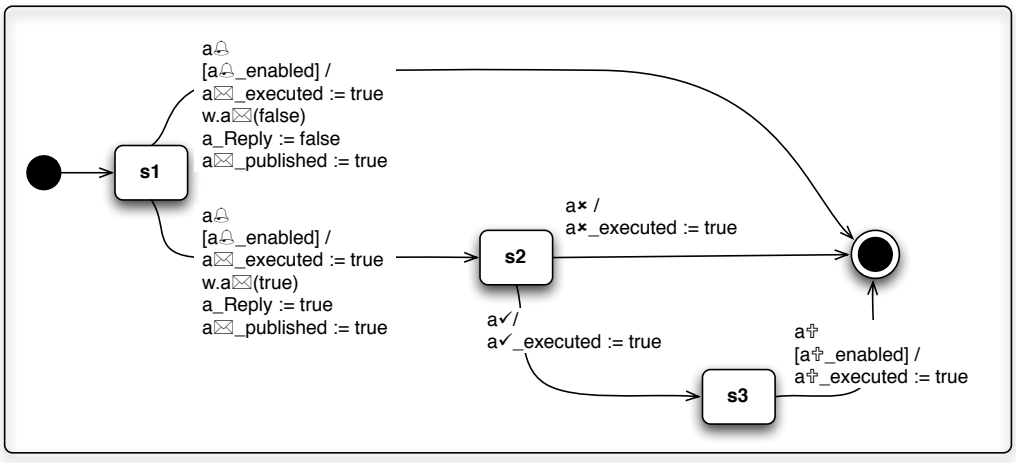
The workflow of the component, given by the UML state declaration SD and transitions T is (a refinement of $ORCH$) assumed to be given a priori.

Requires-interfaces *Let $r \in R$ be a requires-interface, with $spec(r) = \langle \langle NAME, PARAM \rangle, BHV \rangle$. $CLASS(o^r) = \langle SGN, P, VAR, INI, \langle S, E, root, PLL \rangle, T \rangle$ is such that:*

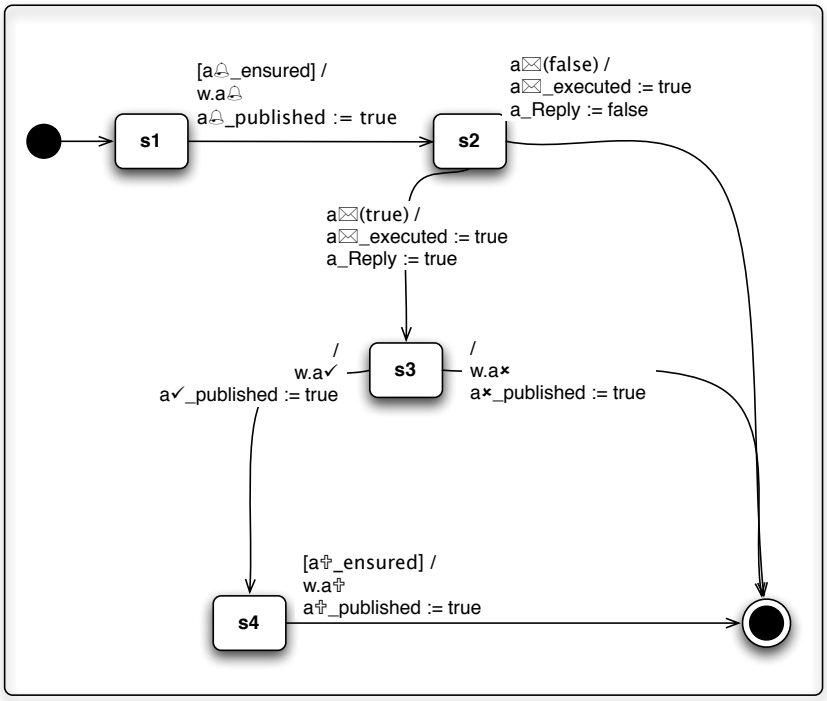
- *$SGN = EN^{RCV}$, i.e. there is a signal for each event that can be received by the required service;*
- *$P(e) = \{p \in PARAM : p \text{ is a parameter of } e\}$ i.e. signals have the same parameters as the corresponding events;*
- *VAR is such that:*

- *For each event $e \in EN^{RCV}$, $e\_enabled, e\_executed \in VAR$, i.e. for each event that can be received by the external service there are two boolean attributes — e_enabled and e_executed — for flagging when the event is enabled for execution and has been executed in the past, respectively;*
- *For each event $e \in EN^{PUB}$, $x\_ensured, x\_published \in VAR$, i.e. for each event that can be published by the external service that are two boolean variables — e_ensured and e_published — for flagging when that the event needs to be published or has been published in the past, respectively;*
- *For each parameter $p \in PARAM(e)$, $e\_p \in VAR$ is a variable used for storing the value of that parameter;*

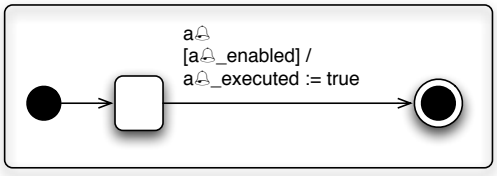
- *$root \in PLL$, i.e. the root state is a composite parallel state;*
- *For each interaction $a \in NAME_{r\&s}$ there is a region in root with the following structure, where $w = o^a$ encodes the wire through which the interaction takes place:*



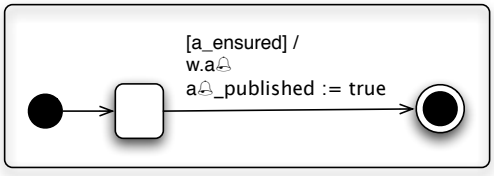
- *For each interaction $a \in NAME_{s\&r}$ there is a region in root with the following structure, where $w = o^a$ encodes the wire through which the interaction takes place:*



- *For each interaction* $a \in NAME_{rcv}$ *there is a region in root with the following structure, where* $w = o^a$ *encodes the wire through which the interaction takes place:*



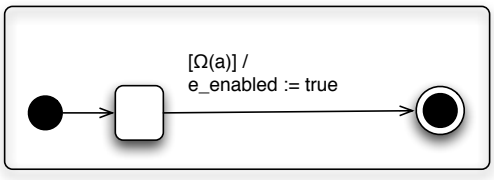
- *For each interaction* $a \in NAME_{snd}$ *there is a region in root with the following structure, where* $w = o^a$ *encodes the wire through which the interaction takes place:*



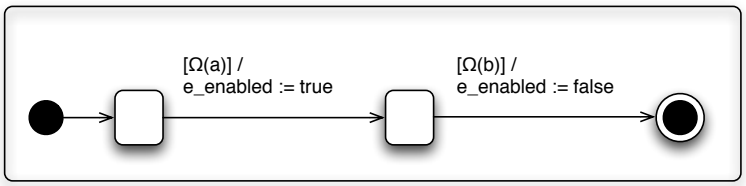
- *For each contraint* $cst \in BHV$ *of type* initiallyEnabled e?, $INI(e\_enabled) = true$, *i.e. the attribute* $e\_enabled$ *is initialized*

to true;

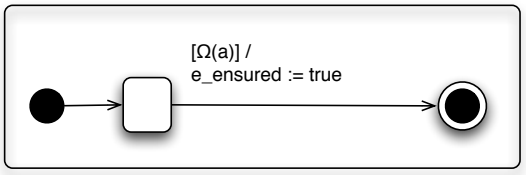
- *For each contraint cst* $\in$ *BHV of type* a enables e? *there is a region in root with the following structure:*



- *For each contraint cst* $\in$ *BHV of type* a enables e? until b *there is a region in root with the following structure:*



- *For each contraint cst* $\in$ *BHV of type* a ensures e? *there is a region in root with the following structure:*



- *For each contraint cst* $\in$ *BHV of type* a.p = c, $INI(a\_p) = c$, *i.e. the attribute* a_p *is initialized to true;*

That is, each interaction defines a region with a workflow pattern that enforces the behaviour associated with that type of interaction; and each constraint (except those defined with the pattern initiallyEnabled*) defines a region with a workflow pattern that enforces that constraint;*

Internal wires *Let* $x \equiv y \in coord$ *(be in the coordination) of some wire* $w = \langle p, p' \rangle \in WW$ *(which connects parties* p *and* p'*), where:*

$spec(w) = \langle map_1, \langle s_1, s_2, coord \rangle, map_2 \rangle$;

$sign(p) = \langle NAME, PARAM \rangle$;

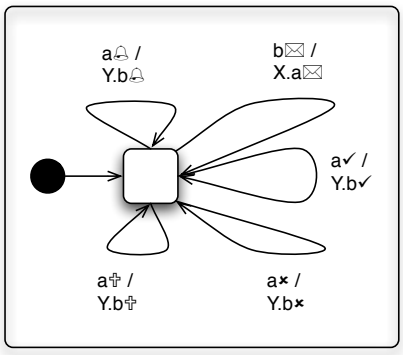
$sign(p') = \langle NAME', PARAM' \rangle$;

$map_1(x) = a$ *for some* $a \in NAME$; *and*

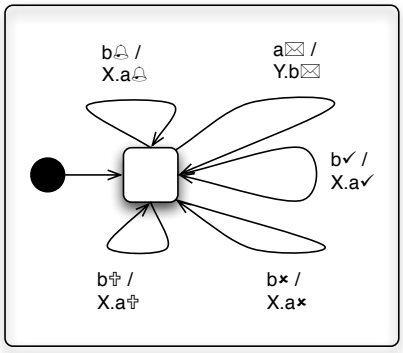
$map_2(y) = b$ *for some* $b \in NAME'$;

$CLASS(o^{x \equiv y}) = \langle SGN, PARAM, VAR, INI, \langle S, E, root, PLL \rangle, T \rangle$ *is such that:*

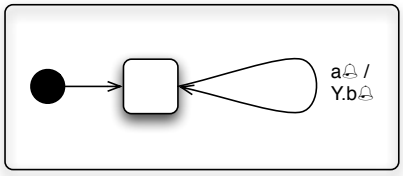
- $SGN = EN_a^{PUB} \cup EN_b'^{PUB}$, *i.e. there is a signal for each event associated with the interaction defined by the equivalence, named according to the party that publishes it;*
- $P(e) = \{p \in PARAM \cup PARAM' : p \text{ is a parameter of } e\}$ *i.e. signals have the same parameters as the corresponding events;*
- $VAR = \emptyset$, *i.e. there are no local attributes;*
- *root is a composite sequential state with the following structure, where* $X = o^p$ *and* $Y = o^{p'}$ *are the objects that encode the two parties connected by the wire:*
  - *If* $a \in NAME_{s\&r}$ *then root is*



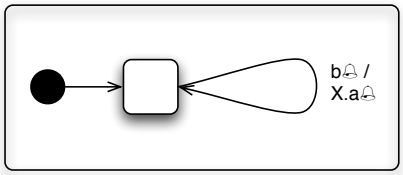
  - *If* $a \in NAME_{r\&s}$ *then root is*



– *If* $a \in NAME_{snd}$ *then root is*



– *If* $a \in NAME_{rcv}$ *then root is*

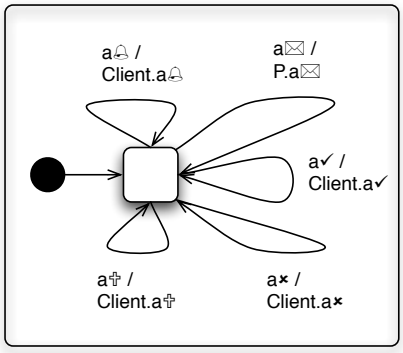


That is, the state machines that encode the internal wires are always ready to forward events from one party to the other using the names that are local to each of the parties;

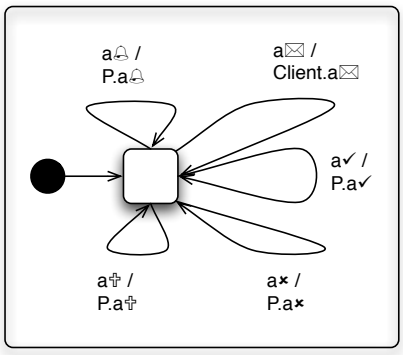
External wires *Let* $prov = \langle\langle NAME, PARAM\rangle_{\_}\rangle$ *be provides-interface of the module and* $a \in NAME$ *be an interaction provided by the service through some party* $p \in P$. $CLASS(o^a) = \langle SGN, P, VAR, INI, \langle S, E, root, PLL\rangle, T\rangle$, *which encodes the wire through which interaction* a *takes place, is such that:*

- $SGN = EN_a$, *i.e. there is a signal for each event associated with interaction* a;
- $P(e) = \{p \in PARAM : p \text{ is a parameter of } e\}$ *i.e. signals have the same parameters as the corresponding events;*

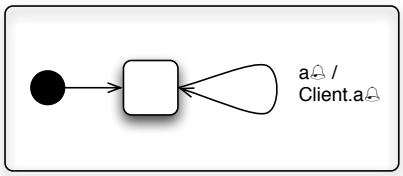
- *VAR* $= \emptyset$, *i.e. there are no local attributes;*
- *root is a composite sequential state with the following structure, where* $P = o^p$ *encodes the party that provides interaction* a*:*
  - *If* $a \in NAME_{s\&r}$ *then*



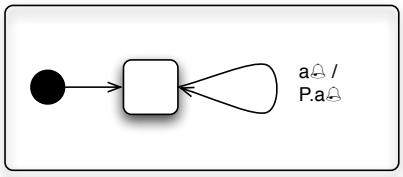
  - *If* $a \in NAME_{r\&s}$ *then*



  - *If* $a \in NAME_{snd}$ *then*



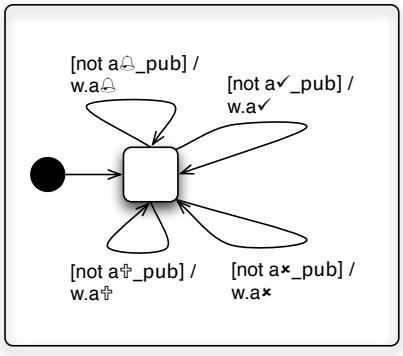
  - *If* $a \in NAME_{rcv}$ *then*



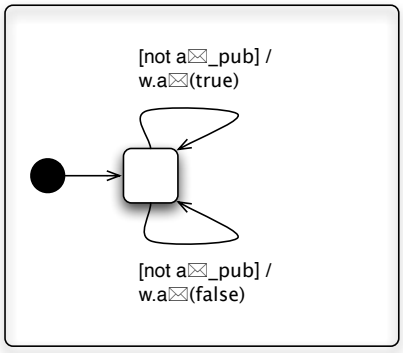
That is, the state machines that encode the external wires are always ready to forward events from the client to the service and vice-versa and do so without changing their names;

Client *Let* $prv = \langle\langle NAME, PARAM\rangle, BHV\rangle$ *be the provides-interface of the module; the class that specifies the behaviour of the client of the service* $CLASS(o^{client}) = \langle SGN, P, VAR, INI, \langle S, E, root, PLL\rangle, T\rangle$ *is such that:*

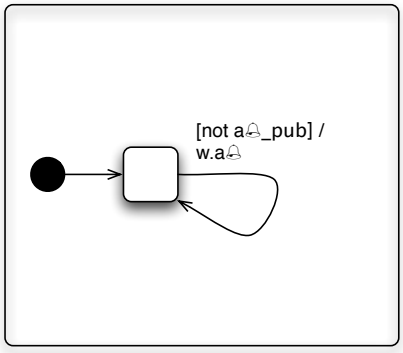
- $SGN = EN^{PUB}$*, i.e. there is a signal for each event that can be published by the service;*
- $P(e) = \{p \in PARAM : p \text{ is a parameter of } e\}$ *i.e. signals have the same parameters as the corresponding events;*
- VAR *is the minimum set such that for each event that can be published by the client (and received by the service), i.e. each event in* EN^{RCV}*, there is an attribute* $e\_published \in VAR$*, which is used for flagging that the event was published and guarantee that each event is published only once;*
- $root$ *is a sequential parallel state such that for every interaction* $a \in NAME \setminus NAME_{snd}$ *(i.e. every interaction that is not typed by snd) there is a region in* $root$ *with the following structure, where* $w = o^a$ *is the wire through which the interaction takes place:*
  - *If* $a \in NAME_{s\&r}$ *then*



  - *If* $a \in NAME_{r\&s}$ *then*



– *If* $a \in NAME_{snd}$ *then*



That is, the state machine that encodes the client can nondeterministically publish the events of each interaction at any given moment, but only publishes each event once.

6.2.3 Model-checking the module *TravelBooking*

In order to model-check that the composition specified by the module *TravelBooking* provides the properties specified in *TravelAgent* (shown in Figure 1.3), we have encoded each of its external-required interfaces, each of the connectors and the client using the methodology described in the previous section.

Having used UMC to model-check *TravelBooking*, we found that all the constraints were satisfied except one: "*payNotify*🔔*!* ∧ *payNotify.status enables bookTrip*✝*?*". This is because there is a path on which the event *bookTrip*✝ is discarded after the event *payNotify*🔔 is published with a positive

value for the *payNotify.status* parameter. This means that the publication of event *payNotify*♠ with a positive *payNotify.status* by the service does not guarantee that the revoke event of interaction *payNotify* becomes enabled for execution. If the composition was implemented as it is, it would be possible for a client to ask for a booking to be revoked and have this request ignored by the service.

After analysing the path that leads to the failure of the property, we understood that the problem is that, because *PA* interacts directly with the client through the wire *CP*, it is possible for the payment notification (represented by *payNotify*♠) to be received by the client before *BA* receives the confirmation for the payment (which is sent via another wire, *BP*). If the client tries to revoke the booking immediately, *BA* will not accept it because it does not yet know that the payment of the booking has been accepted by *PA*.

In order to fix this problem we have redesigned the module *TravelBooking* by removing the wire *CP* and delegating the notification to *BA*. In the new configuration, *PA* does not interact directly with the client anymore. When the payment is executed by *PA*, the component *BA* is notified and is in turn responsible for notifying the client. Only then can the client choose to revoke the booking. Figure 6.4 shows the configuration of the module *TravelBooking* after being redesigned. With its new configuration the module provides all the properties specified in *TravelAgent* (shown in Figure 1.3).

6.2.4 Evaluating the methodology

Completness

Our methodology is used under the assumption that the UML state machines that encode required services and wires satisfy the properties that are specified for those services and wires in the service module (via the associated business and interaction protocols). Under this assumption, we can conclude that each error found by the model-checker reveals a design error in the service module, i.e. it reveals that the module is not correct (in the sense of Def. 5.5.5) — this is illustrated in Section 6.2.3, where the error found by

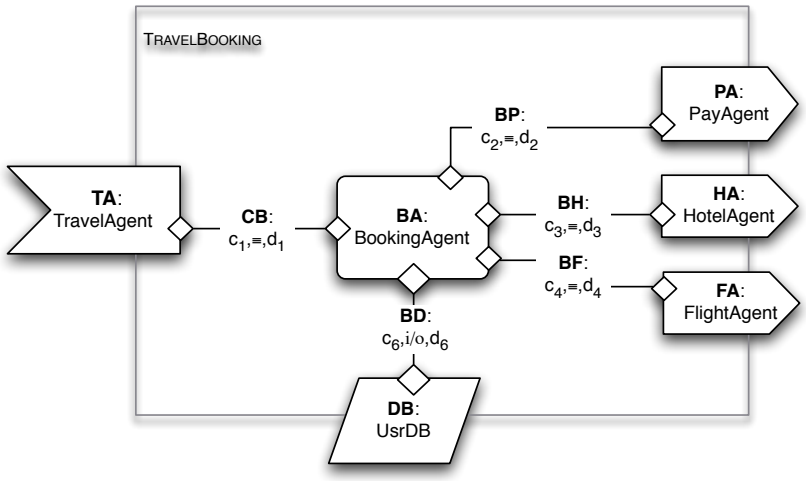


Figure 6.4: The structure of the module *TravelBooking* after being redesigned.

model-checking *TravelBooking* led to redesigning that service module.

On the other hand, if no error is found by the model-checker we cannot conclude that a module is correct. This is because when we use the proposed methodology we restrict the model-checking to a specific selection of required services and wires (those that result from the encoding). Nonetheless, a module is correct only if it provides the intended behaviour for any combination of required services and wires that satisfy the specification in that module.

In summary, this means that while the methodology that we propose can be used for finding design errors — as illustrated in Section 6.2.3 — and therefore indirectly promotes the development of modules that are correct, this methodology does not allow proving that modules are indeed correct. In Chapter 7, we discuss how further work could allow us to prove the correctness of service modules with model-checking.

Performance

We have not made a thorough analysis of the performance of UMC while model-checking SRML modules. We did nonetheless run some experiments while model-checking *TravelBooking* which lead us to the conclusions that we discuss here.

The state space, i.e. the number of states of the L^2TS that is associated

with a service module, can be reduced by making stronger assumptions on the models. For example, we can assume that the client of the service follows the protocol of service requester (in the sense of Def. 3.5.1) in every two-way interaction that it engages in with the module. This assumption reduces the set of possible behaviours of the client and therefore greatly reduces the size of the L^2TS that models the system. If we had made such an assumption when model-checking *TravelBooking* we would still have found the error that we have described Section 6.2.3. Nonetheless, it would be possible for some other errors to be missed — for example, it would be impossible to check that *TravelBooking* discards all the revoke-events that arrive before the corresponding commit-events, as expected from a provider (see Def. 3.5.2).

Our experience using UMC suggests that the gains obtained in terms of speed by making stronger assumptions on the models are of little practical significance when model-checking individual service modules. UMC is capable of verifying a complete model (of a SRML module) in a few seconds and therefore we did not find the need to explore the conditions under which models can be made smaller.

Scalability

As discussed throughout this section, each of the components and required services in a module is encoded by an individual UML state machine for model-checking. It is easy to see that, the size of the models that are used for model-checking is directly proportional to the size of the the modules that they encode. On the other hand, SRML modules tend to be small by nature — typically the properties provided by a module result from the behaviour of a small set of components that are wired to a small set of required services. Because of this, the model-checking methodology that we propose — which builds on the principle that service-oriented architectures should be devoloped in a modular way — doesn't seem to have problems of scalability.

Chapter 7

Conclusions and further work

7.1 A short summary and reflection

In this thesis we have presented our contribution towards the development of SRML — a prototype language being developed within the SENSORIA project for supporting the design of service-oriented computational systems at the business level. SRML distinguishes itself from other formal languages being developed for SOC in two important ways: 1) it is based on a set of complex primitives tailored specifically for modelling the business conversations that occur in SOC; 2) it provides support for modelling the static functional aspects that concern the composition of service behaviour as well as the dynamic aspects that concern the run-time discovery, selection and binding of services.

In this thesis we have focused on the part of SRML that concerns the functional behaviour of services (the dynamic aspects of SRML have been addressed in other publications [38]). We have defined a mathematical model of computation for service-oriented systems that captures the business interactions that take place between the constituents of such systems. Our model captures the conversational nature of business interactions by characterizing them as an asynchronous exchange of correlated messages. In the world of SOC, service providers and their clients continuously engage in structured conversations with the goal of negotiating business deals. Our model captures this and allows services to be characterized by the conversations that they support and the properties of those conversations.

We have given a temporal logic axiomatization of this model using the UCTL logic and we have also defined the declarative languages that SRML provides for specifying the functional behaviour of composite services: the language of business roles for specifying components, the language of business protocols for specifying service interface behaviour and the language of interaction protocols for specifying the coordination performed by the wires that establish the interactions. In particular, the language of business protocols is based on a set of patterns of common service behaviour, which we have formalized as temporal properties using UCTL. This logic based semantics is essential for the mechanism of module composition adopted by SRML to

work — in SRML the business logic of a module relies on the discovery of services (provided by other modules) that satisfy a set of specified properties (see [37, 38]).

In order to analyse the functional correctness of service compositions we have proposed a methodology for using the model-checker UMC that includes encoding the patterns of service behaviour using UML state machines. This methodology is introduced in tune with the goals of SRML (and the SENSORIA project) of supporting service engineering from a practical point of view and has proved itself to be valuable for identifying design errors that could otherwise be overseen.

7.2 The impact of our research

7.2.1 Fine-tuning SRML

In [36], an initial sketch of the syntax of SRML was introduced that had no formal semantics. A main goal of our work has been to formalize that syntax and give it a semantics. One of the side-effects of this process has been the fine-tuning of the language and the clarification of some of its aspects. In particular:

- the language of coordinations, which is used for writing interaction protocols (see Section 5.4), was originally less strict than the current one. The assumptions introduced by the computational model, which resulted from a thorough analysis of the case studies, led us to restrict this language. In the computational model that we have introduced, wires always deliver events to the correct recipient. For example, it is not possible for a party to publish an event and receive it later. The sender and receiver of each event is fixed in the model being considered. Because of this, the purpose of an interaction protocol is not to specify what a wire does with each event, but simply to pair interaction names therefore defining the sender and receiver of the events of those interactions.

- some syntactic redundancy has been eliminated from the language of business roles (see Section 5.2). In [36], we would write $e?$ or $e!$ in the trigger or in the effects of transition specifications, respectively. Our work has clarified that if we write e in the trigger of a transition its meaning is the processing of event e. And if e is written in the effects of a transition then its meaning is the publication of e. In both cases it is not necessary to sufix the name of the event.

- the semantics of provides-interfaces of modules was clarified and as a consequence the notion of module composition that was originally introduced in [37] was updated to that of [38]. As discussed in Section 5.5, a provides-interface defines which subset of interactions associated with the components and requires-interfaces is provided by the module, i.e. it defines for each of the components and required services if and how they can interact with the client of the service. The interaction protocol that the module supports for establishing these interactions can also be part of the specification of the module, but it is not relevant for the properties of those interactions — that is why in this thesis we have only considered modules where the external wires are not specified (see 5.5). The mechanism of module composition described in [37] included the composition of the interaction protocol that a module establishes with its client module and the interaction protocol that the client module was expecting to use (i.e. the one that links to the requires-interface of that module). While defining the semantics of provides-interfaces it became clear that a notion of wire matching should be introduced in the mechanism of module composition, i.e. the interaction protocol that a module supports for interacting with its client module must satisfy (and not be composed with) the interaction protocol that the client module is expecting to use — this is captured in [38].

  From the syntactic point of view, it is important to mention that, before this work was carried out, the interaction signatures associated with the provides-interfaces of modules were typed from the point of view of the

client of those modules. Instead, those signatures must be typed from the point of view of the module (which provides the interactions) in order for the mechanism of matching requires- and provides-interfaces described in [38] to work.

7.2.2 Supporting service-oriented engineering

The model of computation that we have presented in Chapter 3 and the model of service discovery and binding [38] are two fundamental stepping stones for the goals of the SRML approach and project SENSORIA — they provide the mathematical characterization of service-oriented computation that we require for developing formal methodologies for engineering service-oriented systems. Several avenues of research are being pursued that are grounded on these models. Namely:

- Languages and techniques are being pursued for specifying and analysing the functional behaviour of composite services. In this thesis we have defined the three declarative languages that SRML provides for this purpose (the languages of business roles, business protocols and interaction protocols) together with a model-checking technique. Variants and extensions of these languages (in particular the use of state machines and the extension of business protocols with other patterns) are being considered as well as the enrichment of the analytical techniques.

- A complementary approach is being developed for supporting the specification and analysis of service performance using stochastic formalisms [14] — it is at this level that the real-time primitives that we have introduced in this thesis (namely the deadline associated with two-way interactions and the delay associated with wires) are being put to use.

- A methodological approach to the engineering of service-oriented systems has been proposed that advocates capturing requirements using extended versions of UML notation (namely use-case diagrams and statecharts) for deriving (formal) SRML specifications [16].

- An encoding of BPEL into SRML has been defined that provides the means for using BPEL processes as part of composite service specifications; tool support has been developed for automating this encoding process using graph transformations [17].

- The SRML framework is being validated in different use-case scenarios. SRML has been used to model an on-the-road car repair service [15] — an argument is made that service-orientation would enhance the flexibility and adaptability of the computation systems used in the automotive domain. A service for finding a mortgage has been used to illustrate the SRML approach to software engineering [16]. The service for booking a trip that we have used throughout this thesis has also been used to illustrate how the configurations of systems evolve in a service-oriented environment [38].

7.3 Further work

7.3.1 On the specification patterns

The set of interaction types and specification patterns that we have presented is based on the set of case studies that have been adopted in SENSORIA. A more exhaustive survey of case studies will most likely lead us to extend or adapt this set of types and patterns.

We are also interested in making a thorough analysis of the relation between the specification patterns of service interactions and those patterns that are considered essential (if not sufficient) for specifying distributed systems in general [53]. It is our intuition that the former is a subset of the latter. In any case, we believe that SOC would benefit from a circumscribed and well-founded set of patterns for specifying service interactions.

On the model-checking side of things, a formal analysis of the relation between UCTL and UMC state machines should be made if we are to consider extending SRML with more specification patterns. So far, it is not clear the extent to which the (parallel-region-based) compositional approach that we

have proposed for model-checking with UMC can be applied if SRML is extended with other specification patterns.

7.3.2 On the analysis techniques

Validating the current approach

While in UMC one defines an individual model of computation — by inputting a set of UML state machines — in our framework one defines a family of models of computation — by specifying a service module. We would like to prove that the encoding of a service module with UMC state machines, as defined in Chapter 6, satisfies the UCTL formulas that are associated with the requires-interfaces of that module. In order to do this, it is first necessary to formalize the semantics of UMC models, which so far has not been done.

Proving the correctness of service modules with UMC

Ideally, we would like to be able to prove the correctness of a module specification (see Definition 5.5.5), i.e. prove that every possible encoding of the composition specified in that module satisfies the interface properties that the module advertises. In order to be able to perform this task with UMC we would need to address two issues:

- first, we would need to define what a canonical model for a module specification is; intuitively this would be the SO-L^{2}TS that models every possible behaviour of the service module;
- second, we would need to be able to encode that canonical model with a system of UMC state machines, which is not necessarily possible — a formal comparison between the semantic domains of SRML and UMC would have to be done in order to understand exactly which family of SO-L^{2}TS can be encoded with UMC. An alternative to defining a UMC encoding of the canonical model would be to develop a new model-checker — possibly reusing the UMC engine as its basis — tailored for service module specifications, much like it has been done for COWS

specifications [34]. This model-checker should be able to take a service module specification as input together with a UCTL formula and verify if the canonical model associated with that specification verifies that formula.

Model-checking using Linear Temporal Logic

It would be interesting to investigate to which extent the specification patterns that we use in SRML can be formalized with Linear Temporal Logic (LTL). We envision that such a formalization would allow us to use model-checkers like SPIN [1] for supporting service design without having to provide an encoding from temporal properties into UML state machines (unlike the current approach). This would be possible because with LTL it can be expressed that the properties of the requires-interfaces of a service module entail the properties of the provides-interface (over every model that satisfies the behaviour of the components and wires). Intuitively, this consists of checking that in all paths in which the properties of the requires-interfaces are true, the properties of the provides-interfaces are also true — in UCTL (which is as expressive as CTL) this cannot be expressed.

[1] For example, we could use Hugo/RT [64] to translate UML state machine based orchestrations into the modelling language of SPIN, Promela.

Appendix A

Proofs

Let:

- $\Xi = \langle COMP, WIRE, PLL, \Psi, 2WAY, 1WAY \rangle$ be a configuration;
- $m = \langle S, \rightarrow, s_0, G \rangle$ be a model of computation for Ξ;
- $\langle S, s_0, Act, R, L, AP, \Pi \rangle$ be the SO-L^{2}TS that abstracts m.
- $e \in E$, i.e. e is an event associated with Ξ;
- $a \in 2WAY_{\langle c,c' \rangle}$, for some $c, c' \in COMP$, i.e. a is a two-way interaction initiated by c.

A.1 Lemma of sessions

Lemma A.1.1 (Session) *For every two transitions $r, r' \in \rightarrow$ such that $r < r'$, $DLV^r \cap DLV^{r'} = \emptyset$ and $PRC^r \cap PRC^{r'} = \emptyset$, i.e. events are not delivered nor processed more than once on each path.*

Proof

1. *Let $r = s \rightarrow s', r' = s_1 \rightarrow s'_1 \in \rightarrow$ be such that $r < r'$ and $e \in DLV^r$ for some $e \in E$. It is easy to conclude from the notion of computation step (Def. 3.4.2) that there is $r'' < r$ such that $e \in PUB^{r''}$, $e \in PND^s$ and $e \notin PND^{s'}$. According to the notion of session there cannot be $r'' < r'''$ such that $e \in PUB^{r'''}$. Therefore for every $s' \leq s''$, $e \notin PND^{s''}$. Since, $s' \leq s_1$, we conclude from the notion of computation step that $e \notin DLV^{r'}$.*

2. *Let $r = s \rightarrow s', r' = s_1 \rightarrow s'_1 \in \rightarrow$ be such that $r < r'$ and $e \in PRC^r$ for some $e \in E$. It is easy to conclude from the notion of computation step (Def. 3.4.2) that there is $r'' < r$ such that $e \in DLV^{r''}$, $e \in INV^s$ and $e \notin INV^{s'}$. As proved, there cannot be $r'' < r'''$ such that $e \in DLV^{r'''}$. Therefore for every $s' \leq s''$, $e \notin INV^{s''}$. Since, $s' \leq s_1$, we conclude from the notion of computation step that $e \notin PRC^{r'}$.*

3. *The result follows immediately.*

□

A.2 Event propagation (Theorem 4.3.1)

1. We wish to prove that

$$s_0 \models A[true_{\{\neg e!\}} W_{\{e!\}} true]$$

Proof

(a) Let σ be a path from s_0 that contains a transition $s \xrightarrow{\alpha} s'$ with $e! \in \alpha$. In this case, there is $j > 0$, where $\sigma(j-1) = s$, $e! \in \alpha(j)$ and $\sigma(j-1) = s'$. Because $e! \in \alpha(j)$, $\sigma(j-1, j) \models e!$. Obviously $\sigma(j-1) \models true$ and $\sigma(j) \models true$.

Let $0 < i < j$. If $\sigma(i-1, i) \models e¡$ then there is a transition $r = \sigma(i-1) \longrightarrow \sigma(i)$ such that $e \in DLV^r$. Since according to Def. 3.4.2, $DLV^r \subseteq PND^{\sigma(i-1)}$, we conclude that $e \in PND^{\sigma(i-1)}$. We can also conclude from Def. 3.4.2 that either $e! \in PUB^{\sigma(i-2) \longrightarrow \sigma(i-1)}$ or $e! \in PND^{\sigma(i-2)}$. Since, according to Def. 3.4.2 $PND^{s_0} = \emptyset$, it easy to see that there is a transition $r' < r$ such that $e! \in PUB^{r'}$, i.e. there is $'' < i$ such that $e! \in \alpha_{i''}$. But this contradicts Def. 3.4.4 (according to which events can only be published once) since $e! \in \alpha_j$. Hence, $\sigma(i-1, i) \not\models e¡$, i.e. $\sigma(i-1, i) \models \neg e¡$.

Therefore, according to Def. 4.1.6,

$$\sigma \models true_{\{\neg e¡\}} U_{\{e!\}} true$$

(b) Let σ be a path from s_0 that does NOT contain a transition $s \xrightarrow{\alpha} s'$ with $e! \in \alpha$. Then, according to Def 3.4.2, σ cannot contain a

transition $s_1 \xrightarrow{\alpha'} s_1'$ with $e_¡ \in \alpha'$. That is, for all $i \geq 0$, $\sigma(i-1, i) \models \neg e_¡$.

(c) The result follows from (a), (b) and Def. 4.1.6.

□

2. We wish to prove that

$$s_0 \models A[true_{\{\neg e? \wedge \neg e¿\}} W_{\{e¡\}} true]$$

Proof

(a) Let σ be a path from s_0 that contains a transition $s \xrightarrow{\alpha} s'$ with $e_¡ \in \alpha$. In this case, there is $j > 0$, where $\sigma(j-1) = s$, $e_¡ \in \alpha(j)$ and $\sigma(j-1) = s'$. Because $e_¡ \in \alpha(j)$, $\sigma(j-1, j) \models e_¡$. Obviously $\sigma(j-1) \models true$ and $\sigma(j) \models true$.

Let $0 < i < j$. If $\sigma(i-1, i) \models e?$ or $\sigma(i-1, i) \models e¿$ then, there is a transition $r = \sigma(i-1) \longrightarrow \sigma(i)$ such that $e \in EXC^r \cup DSC^r$, i.e. $e \in PRC^r$. Since according to Def. 3.4.2, $DLV^r \subseteq PND^{\sigma(i-1)}$, we conclude that $e \in INV^{\sigma(i-1)}$. We can also conclude from Def. 3.4.2 that either $e! \in DLV^{\sigma(i-2) \longrightarrow \sigma(i-1)}$ or $e! \in INV^{\sigma(i-2)}$. Since, according to Def. 3.4.2 $INV^{s_0} = \emptyset$, it easy to see that there is a transition $r' < r$ such that $e! \in DLV^{r'}$, i.e. there is $'' < i$ such that $e_¡ \in \alpha_{i''}$. But this contradicts Lemma A.1.1 (according to which events can only be delivered once) since $e_¡ \in \alpha_j$. Hence, $\sigma(i-1, i) \not\models e?$ and $\sigma(i-1, i) \not\models e¿$, i.e. $\sigma(i-1, i) \models \neg e? \wedge \neg e¿$.

Therefore, according to Def. 4.1.6,

$$\sigma \models true_{\{\neg e? \wedge \neg e¿\}} U_{\{e¡\}} true$$

(b) Let σ be a path from s_0 that does NOT contain a transition $s \xrightarrow{\alpha} s'$ with $e_¡ \in \alpha$. Then, according to Def 3.4.2, σ cannot contain a

transition $s_1 \xrightarrow{\alpha'} s_1'$ with $e? \in \alpha'$ or $e_{¡} \in \alpha'$. That is, for all $i \geq 0$, $\sigma(i-1, i) \models \neg e? \wedge \neg e_{¿}$.

(c) The result follows from (a), (b) and Def. 4.1.6.

□

3. We wish to prove that

$$s_0 \models AG\neg(e? \wedge e_{¿})$$

Proof Let σ be a path from s_0 and $0 \leq i$.

Let us assume that $e? \in L(\sigma(i))$, i.e. $\sigma(i) \models e?$. From Def. 4.2.1 we conclude that there is a state $s = \sigma(i)$ such that $e \in HST?^s$. From Def. 3.4.2 we conclude that there is a transition $r < s$ such that $e \in EXC^r$ and therefore $e \in PRC^r$. From lemma A.1.1 we conclude that there is no transition r' in σ such that $r' \neq r$ and $PRC^{s' \longrightarrow s''}$. Hence, there is no transition $r' \neq r$ and $DSC^{s' \longrightarrow s''}$. We conclude from 3.4.2 that there is no state s' in σ such that $e \in HST_{¿}$. From 3.4.2 and 4.2.1 we conclude that there is no $0 \leq j$ such that $\sigma \models e_{¿}$. Hence, $\sigma(i) \not\models e_{¿}$ and obviously $\sigma(i) \not\models e? \wedge e_{¿}$, i.e. $\sigma(i) \models \neg(e? \wedge e_{¿})$.

We have concluded that if $\sigma(i) \models e?$ then $\sigma(i) \not\models e_{¿}$. Obviously, if $\sigma(i) \models e_{¿}$ then $\sigma(i) \not\models e?$ and therefore $\sigma(i) \models \neg(e? \wedge e_{¿})$. In the case in which $\sigma(i) \not\models e?$ and $\sigma(i) \not\models e_{¿}$ it is also true that $\sigma(i) \models \neg(e? \wedge e_{¿})$. The result follows immediately from 4.1.7.

□

A.3 Fairness (Theorem 4.3.2)

1. We wish to prove that

$$s_0 \models AG[e!]A[true_{true}U_{e¡}true]$$

Proof Let σ be a path from s_0 and $s = \sigma(i)$ for some $0 \leq i$.

Let $\sigma' = s \xrightarrow{\alpha} s' \ldots$ be such that $e! \in \alpha$, i.e. $e \in PUB^{s \to s'}$. In this case $s \xrightarrow{\alpha} s' \models e!$. We can conclude from Def. 3.4.2 that $e \in PND^{s'}$. According to Def. 3.4.4 (fairness) there is a transition r such that $s' < r$ and $e \in DLV^r$. Since according to Def. 3.4.4 wires are reliable, $DLV^r = ADLV^r$, and therefore there is a $1 < i$ such that $\sigma(i-1, i) \models e¡$. From Def. 4.1.6, we conclude that

$$\sigma' \models true_{\{true\}}U_{\{e¡\}}true$$

and that

$$\sigma'(1) = s' \models A[true_{\{true\}}U_{\{e¡\}}true]$$

We can also conclude from Def. 4.1.7 (the definition of the derived operators) that

$$s \models [e!](A[true_{\{true\}}U_{\{e¡\}}true])$$

The result follows immediately from Def. 4.1.7. □

2. We wish to prove that

$$s_0 \models AG[e¡]A[true_{\{true\}}U_{\{e?\vee e¿\}}true]$$

Proof Let σ be a path from s_0 and $s = \sigma(i)$ for some $0 \leq i$.

Let $\sigma' = s \xrightarrow{\alpha} s' \ldots$ be such that $e¡ \in \alpha$, i.e. $e \in ADLV^{s \to s'}$. In this case $s \xrightarrow{\alpha} s' \models e¡$. Since we assume every wire is reliable (see

Def.3.4.4), we can conclude from Def. 3.4.2 that $e \in INV^{s'}$. According to Def. 3.4.4 (fairness) there is a transition r such that $s' < r$ and $e \in PRC^r$, i.e. $e \in EXC^r \cup DSC^r$. Hence, there is a $1 < i$ such that $\sigma(i-1, i) \models e? \vee e¿$. From Def. 4.1.6, we conclude that

$$\sigma' \models true_{\{true\}} U_{\{e? \vee e¿\}} true$$

and that

$$\sigma'(1) = s' \models A[true_{\{true\}} U_{\{e? \vee e¿\}} true]$$

We can also conclude from Def. 4.1.7 (the definition of the derived operators) that

$$s \models [e¡](A[true_{\{true\}} U_{\{e? \vee e¿\}} true])$$

The result follows immediately from Def. 4.1.7. □

A.4 Session (Theorem 4.3.3)

1. We wish to prove that

$$s_0 \models AG[e!]A[true_{\{\neg e!\}} W false]$$

Proof Let σ be a path from s_0 and $s = \sigma(i)$ for some $0 \leq i$.

Let $\sigma' = (s \xrightarrow{\alpha_1} s')\sigma''$ be such that $e! \in \alpha_1$, i.e. $e \in PUB^{s \to s'}$. In this case $\sigma'(0, 1) \models e!$. According to Def. 3.4.4 (session) for every $s \to s' < r$, $PUB^{s \to s'} \cap PUB^r = \emptyset$, i.e. $e \notin PUB^r$. We can conclude that for every $1 < i$, $\sigma''(i-1, i) \models \neg e!$. From Def. 4.1.6, we conclude that

$$\sigma' \models true_{\{\neg e!\}} W false$$

and that

$$\sigma'(1) = s' \models A[true_{\{\neg e!\}} W false]$$

We can also conclude from Def. 4.1.7 (the definition of the derived operators) that

$$s \models [e!]A[true_{\{\neg e!\}} W false]$$

The result follows immediately from Def. 4.1.7. □

2. We wish to prove that

$$s_0 \models AG[e? \vee e¿]A[true_{\{\neg e? \wedge \neg e¿\}} W false]$$

Proof Let σ be a path from s_0 and $s = \sigma(i)$ for some $0 \leq i$.

Let $\sigma' = (s \xrightarrow{\alpha_1} s')\sigma''$ be such that $e? \in \alpha_1$ or $e¿ \in alpha_1$, i.e. $e \in EXC^{s \to s'} \cup DSC^{s \to s'}$ and therefore $e \in PRC^{s \to s'}$. In this case $\sigma'(0,1) \models e? \vee e¿$. According to Lemma A.1.1 for every $s \to s' < r$, $PRC^{s \to s'} \cap PRC^r = \emptyset$. Hence, $e \notin PRC^r$ and therefore $e \notin EXC^{s \to s'} \cup DSC^{s \to s'}$. We can conclude that for every $1 < i$, $\sigma'(i-1, i) \models \neg e! \wedge \neg e¿$. From Def. 4.1.6, we conclude that

$$\sigma'' \models true_{\{\neg e! \wedge \neg e¿\}} W false$$

and that

$$\sigma'(1) = s' \models A[true_{\{\neg e! \wedge \neg e¿\}} U false]$$

We can also conclude from Def. 4.1.7 (the definition of the derived operators) that

$$s \models [e! \wedge e¿](A[true_{\{\neg e! \wedge \neg e¿\}} U false])$$

The result follows immediately from Def. 4.1.7. □

A.5 Requester (Theorem 4.3.4)

A.5.1 $\Rightarrow$

Let us assume that c behaves as a requester (see Def. 3.5.1) in interaction a.

1. We wish to prove that

$$s_0 \models A[true_{\{\neg a\boxtimes ?\}} W_{\{a\spadesuit !\}} true]$$

Proof

(a) Let σ be a path from s_0 that contains a transition $s \xrightarrow{\alpha} s'$ with $a\spadesuit ! \in \alpha$. In this case, there is $j > 0$, where $\sigma(j-1) = s$, $a\spadesuit ! \in \alpha(j)$ and $\sigma(j-1) = s'$. Because $a\spadesuit ! \in \alpha(j)$, $\sigma(j-1, j) \models a\spadesuit !$. Obviously $\sigma(j-1) \models true$ and $\sigma(j) \models true$.

Let $0 < i < j$. If $\sigma(i-1, i) \models a\boxtimes ?$ then according to Def. 3.5.1 there should be a $i' < i$ such that $\sigma(i'-1, i') \models a\spadesuit !$. But this contradicts Def. 3.4.4 (session). Hence, $\sigma(i-1, i) \not\models a\boxtimes !$, i.e. $\sigma(i-1, i) \models \neg a\boxtimes ?$.

Therefore, according to Def. 4.1.6, $\sigma \models true_{\{\neg a\boxtimes ?\}} U_{\{a\spadesuit !\}} true$.

(b) Let σ be a path from s_0 that does NOT contain a transition $s \xrightarrow{\alpha} s'$ with $a\spadesuit ! \in \alpha$. Then, according to Def. 3.5.1, σ cannot contain a transition $s_1 \xrightarrow{\alpha_1} s_1'$ with $a\boxtimes ? \in \alpha$. That is, for all $i \geq 0$, $\sigma(i-1, i) \models \neg a\boxtimes ?$.

(c) The result follows from (a), (b) and Def. 4.1.6.

□

2. We wish to prove that

$$s_0 \models AG[a\spadesuit !](A[true_{\{\neg a\boxtimes ¿\}} W false])$$

Let $\sigma =$ be a path from s_0 and $s = \sigma(i)$ for some $0 \leq i$.

Let $\sigma' = (s \xrightarrow{\alpha_1} s')\sigma''$ be such that $a\text{🔔}! \in \alpha_1$, i.e. $a\text{🔔} \in PUB^{s \longrightarrow s'}$. In this case $\sigma'(0,1) \models a\text{🔔}?$. According to Def. 3.5.1 there is no transition r such that $s \longrightarrow s' < r$ and $a\text{✉} \in DSC^r$. That is, for every $2 \leq i$, $\sigma'(i-1,i) \models \neg a\text{✉¿}$ and obviously $\sigma'(i) \models true$. From Def. 4.1.6, we conclude that

$$\sigma'' \models true_{\{\neg a.reply¿\}} W false$$

and that

$$\sigma'(1) = s' \models A[true_{\{\neg a.reply¿\}} W false]$$

Since $\alpha_1 \models a\text{🔔}?$, we can also conclude from Def. 4.1.7 (the definition of the derived operators) that

$$s \models [a\text{🔔}!](A[true_{\{\neg a.reply¿\}} W false])$$

The result follows immediately from Def. 4.1.7.

3. We wish to prove that

$$s_0 \models A[true_{\{\neg a\checkmark!\}} W_{\{a\text{✉}?\wedge a.Reply\}} true]$$

Proof

(a) Let σ be a path from s_0 that contains a transition $s \xrightarrow{\alpha} s'$ with $a\text{✉}? \in \alpha$ and $a.Reply^{\Theta^{s \rightarrow s'}}$. In this case, there is $j > 0$, where $\sigma(j-1) = s$, $a\text{✉}? \in \alpha(j)$ and $\sigma(j-1) = s'$. Because $a\text{✉}? \in \alpha(j)$ and $a.Reply^{\Theta^{s \rightarrow s'}}$, $\sigma(j-1,j) \models a\text{✉}? \wedge a.Reply$. Obviously $\sigma(j-1) \models true$ and $\sigma(j) \models true$.

Let $0 < i < j$. If $\sigma(i-1,i) \models a\checkmark!$ then according to Def. 3.5.1 there should be a $i' < i$ such that $\sigma(i'-1,i') \models a\text{✉}? \wedge a.Reply$. But this contradicts Def. 3.4.4 (session). Hence, $\sigma(i-1,i) \not\models a\checkmark!$,

i.e. $\sigma(i-1,i) \models \neg a\checkmark!$.

Therefore, according to Def. 4.1.6, $\sigma \models true_{\{\neg a\checkmark!\}} U_{\{a\boxtimes ?\wedge a.Reply\}} true$.

(b) Let σ be a path from s_0 that does NOT contain a transition $s \xrightarrow{\alpha} s'$ with $a\spadesuit! \in \alpha$. Then, according to Def. 3.5.1, σ cannot contain a transition $s_1 \xrightarrow{\alpha_1} s_1'$ with $a\checkmark! \in \alpha$. That is, for all $i \geq 0$, $\sigma(i-1,i) \models \neg a\checkmark!$.

(c) The result follows from (a), (b) and Def. 4.1.6.

□

4. We wish to prove that

$$s_0 \models AG[a\text{✗}!]A[true_{\{\neg a\checkmark!\}} W false]$$

Proof Let σ be a path from s_0 and $s = \sigma(i)$ for some $0 \leq i$.

Let $\sigma' = (s \xrightarrow{\alpha_1} s')\sigma''$ be such that $a\text{✗}! \in \alpha_1$, i.e. $a\text{✗} \in PUB^{s \to s'}$. In this case $s \xrightarrow{\alpha_1} s' \models a\text{✗}!$. If there is a transition $r < r'$ in σ'' such that $a\checkmark \in PUB^r$, then according to Def. 3.5.2 there cannot be a transition $r'' < r'$ in σ'' such that $a\text{✗} \in PUB^{r''}$. But this contradicts the fact that $s \to s' < r'$ and $a\text{✗} \in PUB^{s \to s'}$. Hence there is no transition r' in σ'' such that $a\checkmark \in PUB^r$. That is, for every $1 < i$, $\sigma(i-1,i) \models \neg a\checkmark!$. From Def. 4.1.6, we conclude that

$$\sigma'' \models true_{\{\neg a\checkmark!\}} W false$$

and that

$$\sigma'(1) = s' \models A[true_{\{\neg a\checkmark!\}} W false]$$

We can also conclude from Def. 4.1.7 (the definition of the derived operators) that

$$s \models [a\text{✗}!]A[true_{\{\neg a\checkmark!\}} W false]$$

The result follows immediately from Def. 4.1.7. □

5. We wish to prove that

$$s_0 \models A[true_{\{\neg a\text{✗}!\}} W_{\{a\boxtimes ?\wedge a.Reply\}} true]$$

Proof

(a) Let σ be a path from s_0 that contains a transition $s \xrightarrow{\alpha} s'$ with $a\boxtimes ? \in \alpha$ and $a.Reply^{\Theta^{s\to s'}}$. In this case, there is $j > 0$, where $\sigma(j-1) = s$, $a\boxtimes ? \in \alpha(j)$ and $\sigma(j-1) = s'$. Because $a\boxtimes ? \in \alpha(j)$ and $a.Reply^{\Theta^{s\to s'}}$, $\sigma(j-1, j) \models a\boxtimes ? \wedge a.Reply$. Obviously $\sigma(j-1) \models true$ and $\sigma(j) \models true$.

Let $0 < i < j$. If $\sigma(i-1, i) \models a\text{✗}!$ then according to Def. 3.5.1 there should be a $i' < i$ such that $\sigma(i'-1, i') \models a\boxtimes ? \wedge a.Reply$. But this contradicts Def. 3.4.4 (session). Hence, $\sigma(i-1, i) \not\models a\text{✗}!$, i.e. $\sigma(i-1, i) \models \neg a\text{✗}!$.

Therefore, according to Def. 4.1.6, $\sigma \models true_{\{\neg a\text{✗}!\}} U_{\{a\boxtimes ?\wedge a.Reply\}} true$.

(b) Let σ be a path from s_0 that does NOT contain a transition $s \xrightarrow{\alpha} s'$ with $a\text{✦}! \in \alpha$. Then, according to Def. 3.5.1, σ cannot contain a transition $s_1 \xrightarrow{\alpha_1} s_1'$ with $a\text{✗}! \in \alpha$. That is, for all $i \geq 0$, $\sigma(i-1, i) \models \neg a\text{✗}!$.

(c) The result follows from (a), (b) and Def. 4.1.6.

□

6. We wish to prove that

$$s_0 \models AG[a\text{✓}!]A[true_{\{\neg a\text{✗}!\}} W false]$$

Proof Let σ be a path from s_0 and $s = \sigma(i)$ for some $0 \leq i$.

Let $\sigma' = (s \xrightarrow{\alpha_1} s')\sigma''$ be such that $a\checkmark! \in \alpha_1$, i.e. $a\checkmark \in PUB^{s \to s'}$. In this case $s \xrightarrow{\alpha_1} s' \models a\checkmark!$. If there is a transition $r < r'$ in σ'' such that $a\text{✗} \in PUB^r$, then according to Def. 3.5.2 there cannot be a transition $r'' < r'$ in σ'' such that $a\checkmark \in PUB^{r''}$. But this contradicts the fact that $s \to s' < r'$ and $a\checkmark \in PUB^{s \to s'}$. Hence there is no transition r' in σ'' such that $a\text{✗} \in PUB^r$. That is, for every $1 < i$, $\sigma(i-1, i) \models \neg a\text{✗}!$. From Def. 4.1.6, we conclude that

$$\sigma'' \models true_{\{\neg a\text{✗}!\}} W false$$

and that

$$\sigma'(1) = s' \models A[true_{\{\neg a\text{✗}!\}} W false]$$

We can also conclude from Def. 4.1.7 (the definition of the derived operators) that

$$s \models [a\checkmark!]A[true_{\{\neg a\text{✗}!\}} W false]$$

The result follows immediately from Def. 4.1.7. □

7. We wish to prove that

$$s_0 \models A[true_{\{\neg a\text{↟}!\}} W_{\{a\checkmark!\}} true]$$

Proof

(a) Let σ be a path from s_0 that contains a transition $s \xrightarrow{\alpha} s'$ with $a\checkmark! \in \alpha$. In this case, there is $j > 0$, where $\sigma(j-1) = s$, $a\checkmark! \in \alpha(j)$ and $\sigma(j-1) = s'$. Because $a\checkmark! \in \alpha(j)$, $\sigma(j-1, j) \models a\checkmark!$. Obviously $\sigma(j-1) \models true$ and $\sigma(j) \models true$.

Let $0 < i < j$. If $\sigma(i-1, i) \models a\text{↟}!$ then according to Def. 3.5.1 there should be a $i' < i$ such that $\sigma(i'-1, i') \models a\checkmark!$. But since

$\sigma(j-1,j) \models a\checkmark!$, Def. 3.4.4 (session) is contradicted. Hence, $\sigma(i-1,i) \not\models a\Rsh!$, i.e. $\sigma(i-1,i) \models \neg a\Rsh?$.

Therefore, according to Def. 4.1.6, $\sigma \models true_{\{\neg a\Rsh!\}}U_{\{a\checkmark!\}}true$.

(b) Let σ be a path from s_0 that does NOT contain a transition $s \xrightarrow{\alpha} s'$ with $a\checkmark! \in \alpha$. Then, according to Def. 3.5.1, σ cannot contain a transition $s_1 \xrightarrow{\alpha_1} s_1'$ with $a\Rsh! \in \alpha$. That is, for all $i \geq 0$, $\sigma(i-1,i) \models \neg a\Rsh!$.

(c) The result follows from (a), (b) and Def. 4.1.6.

□

A.5.2 ⇐

Let us assume that s_0 satisfies every formula in 4.3.4 and $r = s \longrightarrow s'$ is a transition of m.

1. We wish to prove that if $a\boxtimes \in EXC^r$ then there is r' with $r' < r$ such that $a\spadesuit \in PUB^{r'}$

   Proof Let us assume that $a\boxtimes \in EXC^r$ and σ is a path from s_0 that contains r. In this case, there is $i > 0$ such that $\sigma(i-1,i) \models a\boxtimes?$. Since according to 4.3.5, $s_0 \models A[true_{\{\neg a\boxtimes?\}}W_{\{a\spadesuit!\}}true]$ then we conclude that

   $$\sigma \models true_{\{\neg a\boxtimes?\}}W_{\{a\spadesuit!\}}true$$

   Because there is $i > 0$ such that $\sigma(i-1,i) \models a\boxtimes?$, then we also conclude from 4.1.6 that

   $$\sigma \models true_{\{\neg a\boxtimes?\}}U_{\{a\spadesuit!\}}true$$

   i.e. there exists $j \geq 1$ such that $\sigma(j-1,j) \models a\spadesuit!$ and for all i such that $0 < i' < j$, $\sigma(i-1,i) \models \neg a\boxtimes?$. Since $\sigma(i-1,i) \models a\boxtimes?$, we conclude that $j < i$. That is, there is a transition $r' < r$ such that $a\spadesuit \in PUB^{r'}$ (as we wanted to prove). □

2. We wish to prove that if $a\text{♠} \in PUB^r$, then there is no r' with $r < r'$ such that $a\boxtimes \in DSC^{r'}$

   Proof Let us assume that $a\text{♠} \in PUB^r$ and $\sigma = \sigma'(\sigma(i-1) \xrightarrow{\alpha_i} \sigma(i))\sigma''$, where $\sigma_{i-1} = s$ and $\sigma_i = s'$, is a path from s_0. In this case, there is $i > 0$ such that $\sigma(i-1, i) \models a\text{♠}?$. According to 4.3.5

   $$s_0 \models AG[a\text{♠}!](A[true_{a\boxtimes ¿\}} W false])$$

   We conclude from Def. 4.1.6 that

   $$s \models [a\text{♠}!](A[true_{\neg a\boxtimes ¿\}} W false])$$

   Since $s \xrightarrow{\alpha} s' \models a\text{♠}!$, we can conclude that

   $$s' \models A[true_{\neg a\boxtimes ¿\}} W false]$$

   and that

   $$\sigma'' \models true_{\neg a\boxtimes ¿\}} W false$$

   which means that for every $i < j$, $\sigma(j) \models \neg a\boxtimes !$. Hence, there is no transition r' such that $r < r'$ and $a\boxtimes \in DSC^{r'}$. □

3. We wish to prove that if $a\checkmark \in PUB^r$ then there is $r' \leq r$ such that $a\boxtimes \in EXC^{r'}$ and $a.reply^{\Theta^{r'}} = true$

   Proof Let us assume that $a\checkmark \in PUB^r$ and σ is a path from s_0 that contains r. In this case, there is $0 < i$ such that $\sigma(i-1, i) \models a\checkmark !$. Since according to 4.3.5, $s_0 \models A[true_{\{\neg a\checkmark !\}} W_{\{a\boxtimes ?\wedge a.Reply\}} true]$ then we conclude that

   $$\sigma \models true_{\{\neg a\checkmark !\}} W_{\{a\boxtimes ?\wedge a.Reply\}} true$$

   Because there is $0 < i$ such that $\sigma(i-1, i) \models a\checkmark !$, it is not the case that for every $0 < i'$, $\sigma(i'-1, i) \models \neg a\checkmark !$. We conclude from 4.1.6 that

   $$\sigma \models true_{\{\neg a\checkmark !\}} U_{\{a\boxtimes ?\wedge a.Reply\}} true$$

i.e. there exists $j \geq 1$ such that $\sigma(j-1, j) \models a\boxtimes? \wedge a.Reply$ and for all i such that $0 < i' < j$, $\sigma(i'-1, i') \models \neg a\checkmark!$. Since $\sigma(i-1, i) \models a\checkmark!$, we conclude that $j < i$. That is, there is a transition $r' < r$ such that $a\boxtimes \in EXC^{r'}$ and $a.reply^{\Theta^{r'}} = true$ (as we wanted to prove). □

4. We wish to prove that if $a\checkmark \in PUB^r$ then there is no $r' < r$ such that $a\text{✗} \in PUB^{r'}$

   Proof Let us assume that $a\checkmark \in PUB^r$ and σ is a path from s_0 that contains r. In this case, there is $i > 0$ such that $\sigma(i-1, i) \models a\checkmark!$. Let us suppose that $\sigma = \sigma'(s_1 \xrightarrow{\alpha'} s_1')\sigma''$ where $r' = s_1 \rightarrow s_1' < r$ and $a\text{✗} \in PUB^{r'}$. In this case $s_1 \xrightarrow{\alpha'} s_1' \models a\text{✗}!$. Since according to 4.3.5

   $$s_0 \models AG[a\text{✗}!](A[true_{\{\neg a\checkmark\}}Wfalse])$$

   we conclude from Def. 4.1.7 that

   $$s_1 \models [a\text{✗}!](A[true_{\{\neg a\checkmark!\}}Wfalse])$$

   Since $s_1 \xrightarrow{\alpha'} s_1' \models a\checkmark!$, we can conclude that

   $$s_1' \models A[true_{\{\neg a\checkmark!\}}Wfalse]$$

   and that

   $$\sigma'' \models true_{\{\neg a\checkmark!\}}Wfalse$$

   which means that for every $i < j$, $\sigma(j) \models \neg a\checkmark!$, i.e. $a\checkmark! \notin \alpha_j$. But this contradicts the fact that $r' < r$ and $r \models a\checkmark!$. Hence, $a\text{✗} \notin PUB^{r'}$ or $r' \not< r$. Therefore, there is no transition $r' < r$ in σ such that $a\text{✗} \in PUB^{r'}$. □

5. We wish to prove that if $a\text{✗} \in PUB^r$ then there is $r' < r$ such that $a\boxtimes \in EXC^{r'}$ and $a.reply^{\Theta^{r'}} = true$

   Proof Let us assume that $a\text{✗} \in PUB^r$ and σ is a path from s_0 that

contains r. In this case, there is $0 < i$ such that $\sigma(i-1, i) \models a\text{✗}!$. Since according to 4.3.5, $s_0 \models A[true_{\{\neg a\text{✗}!\}} W_{\{a\boxtimes ? \wedge a.Reply\}} true]$ then we conclude that

$$\sigma \models true_{\{\neg a\text{✗}!\}} W_{\{a\boxtimes ? \wedge a.Reply\}} true$$

Because there is $0 < i$ such that $\sigma(i-1, i) \models a\text{✗}!$, it is not the case that for every $0 < i'$, $\sigma(i'-1, i) \models \neg a\text{✗}!$. We conclude from 4.1.6 that

$$\sigma \models true_{\{\neg a\text{✗}!\}} U_{\{a\boxtimes ? \wedge a.Reply\}} true$$

i.e. there exists $j \geq 1$ such that $\sigma(j-1, j) \models a\boxtimes ? \wedge a.Reply$ and for all i such that $0 < i' < j$, $\sigma(i'-1, i') \models \neg a\text{✗}!$. Since $\sigma(i-1, i) \models a\text{✗}!$, we conclude that $j < i$. That is, there is a transition $r' < r$ such that $a\boxtimes \in EXC^{r'}$ and $a.reply^{\Theta^{r'}} = true$ (as we wanted to prove). □

6. We wish to prove that if $a\text{✗} \in PUB^r$ then there is no $r' < r$ such that $a\checkmark \in PUB^{r'}$

   Proof Let us assume that $a\text{✗} \in PUB^r$ and σ is a path from s_0 that contains r. In this case, there is $i > 0$ such that $\sigma(s_{i-1}, s_i) \models a\text{✗}!$. Let us suppose that $\sigma = \sigma'(s_1 \xrightarrow{\alpha'} s'_1)\sigma''$ where $r' = s_1 \rightarrow s'_1 < r$ and $a\checkmark \in PUB^{r'}$. In this case $s_1 \xrightarrow{\alpha'} s'_1 \models a\checkmark!$. Since according to 4.3.5

$$s_0 \models AG[a\text{✗}!](A[true_{\{\neg a\text{✗}\}} W false])$$

   we conclude from Def. 4.1.7 that

$$s_1 \models [a\checkmark!](A[true_{\{\neg a\text{✗}!\}} W false])$$

   Since $s_1 \xrightarrow{\alpha'} s'_1 \models a\text{✗}!$, we can conclude that

$$s'_1 \models A[true_{\{\neg a\text{✗}!\}} W false]$$

   and that

$$\sigma'' \models true_{\{\neg a\text{✗}!\}} W false$$

which means that for every $i < j$, $\sigma(j) \models \neg a\text{✗}!$, i.e. $a\text{✗}! \notin \alpha_j$. But this contradicts the fact that $r' < r$ and $r \models a\text{✗}!$. Hence, $a\checkmark \notin PUB^{r'}$ or $r' \not< r$. Therefore, there is no transition $r' < r$ in σ such that $a\checkmark \in PUB^{r'}$. □

7. We wish to prove that if $a\text{✞} \in PUB^r$ then there is $r' < r$ such that $a\checkmark \in PUB^{r'}$

   Proof Let us assume that $a\text{✞} \in PUB^r$ and σ is a path from s_0 that contains r. In this case, there is $0 < i$ such that $\sigma(i-1, i) \models a\text{✞}!$. Since according to 4.3.5, $s_0 \models A[true_{\{\neg a\text{✞}!\}} W_{\{a\checkmark!\}} true]$ then we conclude that

   $$\sigma \models true_{\{\neg a\text{✞}!\}} W_{\{a\checkmark!\}} true$$

   Because there is $0 < i$ such that $\sigma(i-1, i) \models a\checkmark!$, it is not the case that for every $0 < i'$, $\sigma(i'-1, i) \models \neg a\checkmark!$. We conclude from 4.1.6 that

   $$\sigma \models true_{\{\neg a\text{✞}!\}} U_{\{a\checkmark!\}} true$$

   i.e. there exists $j \geq 1$ such that $\sigma(j-1, j) \models a\checkmark?$ and for all i such that $0 < i' < j$, $\sigma(i'-1, i') \models \neg a\text{✞}!$. Since $\sigma(i-1, i) \models a\text{✞}!$, we conclude that $j < i$. That is, there is a transition $r' < r$ such that $a\checkmark \in EXC^{r'}$ (as we wanted to prove). □

A.6 Provider (Theorem 4.3.5)

A.6.1 ⇒

Let us assume that c' behaves as a provider (see Def. 3.5.2) in interaction a.

1. We wish to prove that

   $$s_0 \models A[true_{\{\neg a\text{✉}!\}} W_{\{a\text{♞}?\}} true]$$

Proof

(a) Let σ be a path from s_0 that contains a transition $s \xrightarrow{\alpha} s'$ with $a♠? \in \alpha$. In this case, there is $j > 0$, where $\sigma(j-1) = s$, $a♠? \in \alpha(j)$ and $\sigma(j-1) = s'$. Because $a♠? \in \alpha(j)$, $\sigma(j-1, j) \models a♠?$. Obviously $\sigma(j-1) \models true$ and $\sigma(j) \models true$.

Let $0 < i < j$. If $\sigma(i-1, i) \models a✉!$ then by clause 1. in Def. 3.5.2 there should be a $i' < i$ such that $\sigma(i'-1, i') \models a♠?$. But this contradicts lemma A.1.1 (according to which events can only be processed once). Hence, $\sigma(i-1, i) \not\models a✉!$, i.e. $\sigma(i-1, i) \models \neg a✉!$.

Therefore, according to Def. 4.1.6, $\sigma \models true_{\{\neg a✉!\}} U_{\{a♠?\}} true$.

(b) Let σ be a path from s_0 that does NOT contain a transition $s \xrightarrow{\alpha} s'$ with $a♠? \in \alpha$. Then, according to clause 1. in Def. 3.5.2 σ cannot contain a transition $s_1 \xrightarrow{\alpha_1} s_1'$ with $a✉! \in \alpha$. That is, for all $i \geq 0$, $\sigma(i-1, i) \models \neg a✉!$.

(c) The result follows from (a), (b) and Def. 4.1.6.

□

2. We wish to prove that

$$s_0 \models AG[a♠?](A[true_{\{true\}} U_{\{a✉!\}} true])$$

Proof Let σ be a path from s_0 and $s = \sigma(i)$ for some $0 \leq i$.

Let $\sigma' = s \xrightarrow{\alpha} s' \ldots$ be such that $a♠? \in \alpha$. In this case $s \xrightarrow{\alpha} s' \models a♠?$. According to the definition of provider (Def. 3.5.2) there is a transition r such that $s \xrightarrow{\alpha} s' < r$ and $a✉ \in PUB^r$. That is, there is a $1 < i$ such that $\sigma(i-1, i) \models a✉!$. From Def. 4.1.6, we conclude that

$$\sigma' \models true_{\{true\}} U_{\{a✉!\}} true$$

and that

$$\sigma'(1) = s' \models A[true_{\{true\}} U_{\{a\boxtimes !\}} true]$$

We can also conclude from Def. 4.1.7 (the definition of the derived operators) that

$$s \models [a\text{♞}?](A[true_{\{true\}} U_{\{a\boxtimes !\}} true])$$

The result follows immediately from Def. 4.1.7. □

3. We wish to prove that

$$s_0 \models A[true_{\{\neg a\checkmark ?\}} W_{\{a\boxtimes !\wedge a.Reply\}} A[true_{\{\neg a\checkmark ?\}} W_{\{a\checkmark ?\wedge time<a.useBy\}} true]]$$

Proof

(a) Let σ be a path from s_0 that contains a transition $r = s \xrightarrow{\alpha} s'$ such that $a\boxtimes ! \in \alpha$ and $a.Reply^{\Theta^r} = true$. In this case, $\sigma = \sigma'(s, \alpha, s')\sigma''$ and there is a $j > 0$, where $\sigma(j-1) = s$, $a\boxtimes ! \in \alpha(j)$ and $\sigma(j) = s'$. Because $a\boxtimes ! \in \alpha_j$ and $a.Reply^{\Theta^r} = true$, $\sigma(j-1, j) \models a\boxtimes ! \wedge a.Reply$.

i. Let us assume there is a transition r' in σ'' such that $r < r'$ and $a\checkmark \in EXC^{r'}$. In this case there is $0 < i$ such that $\sigma''(i-1, i) \models a\checkmark ?$. From Def. 3.5.2 we conclude that $TIME^{\sigma(i')} < a.useBy^{\Theta^r}$ and therefore, $\sigma(i'-1, i') \models a\checkmark ?\wedge time < a.useBy$. From lemma A.1.1 we conclude that there is no $i' < i$ such that $\sigma(i'-1, i') \models a\checkmark ?$. Hence, from Def. 4.1.6 we conclude that

$$\sigma'' \models true_{\{\neg a\checkmark ?\}} U_{\{a\checkmark ?\wedge time<a.useBy\}} true$$

From lemma A.1.1 we conclude that there is no $i'' < j < i$ such that $\sigma(i''-1, i'') \models a\checkmark ?$, i.e. for every $i'' < j$, $\sigma(i''-1, i'') \models \neg a\checkmark$.

ii. Let us instead assume that there is no transition r' in σ'' such that $r < r'$ and $a\checkmark \in EXC^{r'}$. In this case for every $0 < i$ $\sigma''(i-1, i) \models \neg a\checkmark?$. If we assume that there is $r'' < r$ such that $a\checkmark \in EXC^{r'}$ then according to Def. 3.5.2 there should be $r''' < r''$ such that $a\boxtimes \in PUB^{r'''}$. But since $a\boxtimes \in PUB^{r}$, this is a contradiction with lemma A.1.1. Therefore, there is no $r'' < r$ such that $a\checkmark \in EXC^{r'}$, i.e. for every $0 < i'' < j$, $\sigma(i'-1, i') \models \neg a\checkmark?$.

iii. We conclude from i., ii. and 4.1.6 that

$$\sigma'' \models true_{\{\neg a\checkmark?\}} W_{\{a\checkmark? \wedge time < a.useBy\}} true$$

and that

$$s' \models A[true_{\{\neg a\checkmark?\}} W_{\{a\checkmark? \wedge time < a.useBy\}} true]$$

Furthermore, since $r \models a\boxtimes! \wedge a.Reply$, we conclude that

$$\sigma \models true_{\{\neg a\checkmark?\}} U_{\{a\boxtimes! \wedge a.Reply\}} A[true_{\{\neg a\checkmark?\}} W_{\{a\checkmark? \wedge time < a.useBy\}} true]$$

and that

$$s_0 \models A[true_{\{\neg a\checkmark?\}} U_{\{a\boxtimes! \wedge a.Reply\}} A[true_{\{\neg a\checkmark?\}} W_{\{a\checkmark? \wedge time < a.useBy\}} true]]$$

(b) Let σ be a path from s_0 that does NOT contain a transition $r = s \xrightarrow{\alpha} s'$ such that $a\boxtimes! \in \alpha$ and $a.Reply^{\Theta^r} = true$. According to Def. 3.5.2 there cannot be a transition r' in σ such that $a\checkmark \in EXC^{r'}$, i.e. for every $0 < i$, $\sigma(i-1, i) \models \neg a\checkmark?$.

(c) The result follows from (a), (b) and Def. 4.1.6.

□

4. We wish to prove that

$$s_0 \models AG[a\text{✗}?]A[true_{\{\neg a\text{✓}?\}}Wfalse]$$

Proof Let σ be a path from s_0 and $s = \sigma(i)$ for some $0 \leq i$.

Let $\sigma' = (s \xrightarrow{\alpha_1} s')\sigma''$ be such that $a\text{✗}? \in \alpha_1$, i.e. $a\text{✗} \in EXC^{s \to s'}$. In this case $\sigma'(0,1) \models a\text{✗}?$. If there is a transition r' in σ'' such that $a\text{✓} \in EXC^r$, then according to Def. 3.5.2 there cannot be a transition $r'' < r'$ in σ'' such that $a\text{✗} \in EXC^{r''}$. But this contradicts the fact that $s \to s' < r'$ and $a\text{✗} \in EXC^{s \to s'}$. Hence there is no transition r' in σ'' such that $a\text{✓} \in EXC^r$. That is, for every $0 < i$, $\sigma''(i-1, i) \models \neg a\text{✓}?$. From Def. 4.1.6, we conclude that

$$\sigma'' \models true_{\{\neg a\text{✓}?\}}Wfalse$$

and that

$$\sigma'(1) = s' \models A[true_{\{\neg a\text{✓}?\}}Wfalse]$$

We can also conclude from Def. 4.1.7 (the definition of the derived operators) that

$$s \models [a\text{✗}?]A[true_{\{\neg a\text{✓}?\}}Wfalse]$$

The result follows immediately from Def. 4.1.7. □

5. We wish to prove that

$$s_0 \models A[true_{\{\neg(a\text{✗}?)\}}W_{\{a\text{✉}!\wedge a.Reply\}}A[true_{\{\neg a\text{✗}?\}}W_{\{a\text{✗}?\wedge time<a.useBy\}}true]]$$

Proof

(a) Let σ be a path from s_0 that contains a transition $r = s \xrightarrow{\alpha} s'$ such that $a\text{✉}! \in \alpha$ and $a.Reply^{\Theta^r} = true$. In this case, $\sigma = \sigma'(s, \alpha, s')\sigma''$ and there is a $j > 0$, where $\sigma(j-1) = s$, $a\text{✉}! \in$

$\alpha(j)$ and $\sigma(j) = s'$. Because $a\boxtimes! \in \alpha_j$ and $a.Reply^{\Theta^r} = true$, $\sigma(j-1, j) \models a\boxtimes! \wedge a.Reply$.

i. Let us assume there is a transition r' in σ'' such that $r < r'$ and $a\not\boxtimes \in EXC^{r'}$. In this case there is $0 < i$ such that $\sigma''(i-1, i) \models a\not\boxtimes?$. From Def. 3.5.2 we conclude that $TIME^{\sigma(i')} < a.useBy^{\Theta^r}$ and therefore, $\sigma(i'-1, i') \models a\not\boxtimes? \wedge time < a.useBy$. From lemma A.1.1 we conclude that there is no $i' < i$ such that $\sigma(i'-1, i') \models a\not\boxtimes?$. Hence, from Def. 4.1.6 we conclude that

$$\sigma'' \models true_{\{\neg a\not\boxtimes?\}} U_{\{a\not\boxtimes? \wedge time < a.useBy\}} true$$

From lemma A.1.1 we conclude that there is no $i'' < j < i$ such that $\sigma(i''-1, i'') \models a\not\boxtimes?$, i.e. for every $i'' < j$, $\sigma(i''-1, i'') \models \neg a\not\boxtimes$.

ii. Let us instead assume that there is no transition r' in σ'' such that $r < r'$ and $a\not\boxtimes \in EXC^{r'}$. In this case for every $0 < i$ $\sigma''(i-1, i) \models \neg a\not\boxtimes?$. If we assume that there is $r'' < r$ such that $a\not\boxtimes \in EXC^{r'}$ then according to Def. 3.5.2 there should be $r''' < r''$ such that $a\boxtimes \in PUB^{r'''}$. But since $a\boxtimes \in PUB^r$, this is a contradiction with lemma A.1.1. Therefore, there is no $r'' < r$ such that $a\not\boxtimes \in EXC^{r'}$, i.e. for every $0 < i'' < j$, $\sigma(i'-1, i') \models \neg a\not\boxtimes?$.

iii. We conclude from i., ii. and 4.1.6 that

$$\sigma'' \models true_{\{\neg a\not\checkmark?\}} W_{\{a\not\boxtimes? \wedge time < a.useBy\}} true$$

and that

$$s' \models A[true_{\{\neg a\not\checkmark?\}} W_{\{a\not\boxtimes? \wedge time < a.useBy\}} true]$$

Furthermore, since $r \models a\boxtimes! \wedge a.Reply$, we conclude that

$$\sigma \models true_{\{\neg a\not\boxtimes?\}} U_{\{a\boxtimes! \wedge a.Reply\}} A[true_{\{\neg a\not\boxtimes?\}} W_{\{a\not\boxtimes? \wedge time < a.useBy\}} true]$$

and that

$$s_0 \models A[true_{\{\neg a\text{✗}?\}} U_{\{a\boxtimes !\wedge a.Reply\}} A[true_{\{\neg a\text{✗}?\}} W_{\{a\text{✗}?\wedge time<a.useBy\}} true]]$$

(b) Let σ be a path from s_0 that does NOT contain a transition $r = s \xrightarrow{\alpha} s'$ such that $a\boxtimes ! \in \alpha$ and $a.Reply^{\Theta^r} = true$. According to Def. 3.5.2 there cannot be a transition r' in σ such that $a\text{✗} \in EXC^{r'}$, i.e. for every $0 < i$, $\sigma(i-1, i) \models \neg a\text{✗}?$.

(c) The result follows from (a), (b) and Def. 4.1.6.

□

6. We wish to prove that

$$s_0 \models AG[a\checkmark?]A[true_{\{\neg a\text{✗}?\}} W false]$$

Proof Let σ be a path from s_0 and $s = \sigma(i)$ for some $0 \leq i$.

Let $\sigma' = (s \xrightarrow{\alpha_1} s')\sigma''$ be such that $a\checkmark? \in \alpha_1$, i.e. $a\checkmark \in EXC^{s\rightarrow s'}$. In this case $\sigma'(0,1) \models a\text{✗}?$. If there is a transition r' in σ'' such that $a\text{✗} \in EXC^r$, then according to Def. 3.5.2 there cannot be a transition $r'' < r'$ in σ'' such that $a\checkmark \in EXC^{r''}$. But this contradicts the fact that $s \rightarrow s' < r'$ and $a\checkmark \in EXC^{s\rightarrow s'}$. Hence there is no transition r' in σ'' such that $a\text{✗} \in EXC^r$. That is, for every $1 < i$, $\sigma''(i-1, i) \models \neg a\text{✗}?$. From Def. 4.1.6, we conclude that

$$\sigma'' \models true_{\{\neg a\text{✗}?\}} W false$$

and that

$$\sigma'(1) = s' \models A[true_{\{\neg a\text{✗}?\}} W false]$$

We can also conclude from Def. 4.1.7 (the definition of the derived operators) that

$$s \models [a\checkmark?]A[true_{\{\neg a\text{✗}?\}} W false]$$

The result follows immediately from Def. 4.1.7. □

7. We wish to prove that

$$s_0 \models A[true_{\{\neg(a\boxtimes!\wedge a.Reply)\}} W_{\{a\boxtimes!\wedge a.Reply\}} \neg E[true_{\{\neg a\text{✗}?\}} U_{\{a\checkmark ¿\wedge time<a.useBy\}} true]]$$

Proof

(a) Let σ be a path from s_0 that contains a transition $r = s \xrightarrow{\alpha} s'$ such that $a\boxtimes! \in \alpha$ and $a.Reply^{\Theta^r} = true$. In this case, $\sigma = \sigma'(s, \alpha, s')\sigma''$ and there is a $j > 0$, where $\sigma(j-1) = s$, $a\boxtimes! \in \alpha(j)$ and $\sigma(j) = s'$. Because $a\boxtimes! \in \alpha_j$ and $a.Reply^{\Theta^r} = true$, $\sigma(j-1, j) \models a\boxtimes! \wedge a.Reply$.

According to the definition of provider there is no $r' = n_1 \longrightarrow n_2$ with $r < r'$ such that:

- $a\checkmark \in DSC^{r'}$
- $TIME^{n_1} < a.useBy^{\Theta^r}$
- there is no transition $r'' < r'$ such that $a\text{✗} \in EXC^{r''}$

That is, there is no $j \leq i$ such that $\sigma'' = \sigma_1(\sigma(i-1, i))\sigma_1'$, $\sigma(i-1, i) \models a\checkmark ¿\wedge(time < a.useBy)$ and for all $j \leq i' < i$, $\sigma(i'-1, i') \models \neg a\text{✗}?$. From Def. 4.1.6, we conclude that

$$\sigma'' \not\models true_{\{\neg a\text{✗}?\}} U_{\{a\checkmark ¿\wedge time<a.useBy\}} true$$

Therefore, according to Def. 4.1.6,

$$s' \models \neg E[true_{\{\neg a\text{✗}?\}} U_{\{a\checkmark ¿\wedge time<a.useBy\}} true]$$

Since $a\boxtimes \in PUB^r$ and m is a session (see Def. 3.4.4), there is no transition $r''' < r$ such that $a\boxtimes \in PUB^{r'''}$. That is, for all

$0 < i'' < j$, $\sigma(i''-1, i'') \models \neg(a\boxtimes! \wedge a.Reply)$. Hence,

$$\sigma \models true_{\{\neg(a\boxtimes!\wedge a.Reply)\}} U_{\{a\boxtimes!\wedge a.Reply\}} \neg E[true_{\{\neg a\text{✗}?\}} U_{\{a\checkmark ¿\wedge time<a.useBy\}} true]$$

(b) Let σ be a path from s_0 that does NOT contain a transition $r = s \xrightarrow{\alpha} s'$ with $a\boxtimes! \in \alpha$ and $a.Reply^{\Theta^r} = true$. In this case, for all $i \geq 0$, $\sigma(i-1, i) \models \neg(a\boxtimes! \wedge a.Reply)$.

(c) The result follows from (a), (b) and Def. 4.1.6.

□

8. We wish to prove that

$$s_0 \models A[true_{\{a\boxtimes!\wedge a.Reply\}} W_{\{a\boxtimes!\wedge a.Reply\}} \neg E[true_{\{\neg a\checkmark?\}} U_{\{a\text{✗}¿\wedge time<a.useBy\}} true]]$$

Proof

(a) Let σ be a path from s_0 that contains a transition $r = s \xrightarrow{\alpha} s'$ such that $a\boxtimes! \in \alpha$ and $a.Reply^{\Theta^r} = true$. In this case, $\sigma = \sigma'(s, \alpha, s')\sigma''$ and there is a $j > 0$, where $\sigma(j-1) = s$, $a\boxtimes! \in \alpha(j)$ and $\sigma(j) = s'$. Because $a\boxtimes! \in \alpha_j$ and $a.Reply^{\Theta^r} = true$, $\sigma(j-1, j) \models a\boxtimes! \wedge a.Reply$.

According to the definition of provider there is no $r' = n_1 \longrightarrow n_2$ with $r < r'$ such that:

- $a\text{✗} \in DSC^{r'}$
- $TIME^{n_1} < a.useBy^{\Theta^r}$
- there is no transition $r'' < r'$ such that $a\checkmark \in EXC^{r''}$

That is, there is no $j \leq i$ such that $\sigma'' = \sigma_1(\sigma(i-1, i))\sigma_1'$, $\sigma(i-1, i) \models a\text{✗}¿ \wedge (time < a.useBy)$ and for all $j \leq i' < i$, $\sigma(i'-1, i') \models \neg a\checkmark?$. From Def. 4.1.6, we conclude that

$$\sigma'' \not\models true_{\{\neg a\checkmark?\}} U_{\{a\text{✗}¿\wedge time<a.useBy\}} true$$

Therefore, according to Def. 4.1.6,

$$s' \models \neg E[true_{\{\neg a\checkmark ?\}} U_{\{a\text{✗}¿\wedge time<a.useBy\}} true]$$

Since $a\boxtimes \in PUB^r$ and m is a session (see Def. 3.4.4), there is no transition $r''' < r$ such that $a\boxtimes \in PUB^{r'''}$. That is, for all $0 < i'' < j$, $\sigma(i''-1, i'') \models \neg(a\boxtimes ! \wedge a.Reply)$. Hence,

$$\sigma \models true_{\{\neg(a\boxtimes !\wedge a.Reply)\}} U_{\{a\boxtimes !\wedge a.Reply\}} \neg E[true_{\{\neg a\checkmark ?\}} U_{\{a\text{✗}¿\wedge time<a.useBy\}} true]$$

(b) Let σ be a path from s_0 that does NOT contain a transition $r = s \xrightarrow{\alpha} s'$ with $a\boxtimes ! \in \alpha$ and $a.Reply^{\Theta^r} = true$. In this case, for all $i \geq 0$, $\sigma(i-1, i) \models \neg(a\boxtimes ! \wedge a.Reply)$.

(c) The result follows from (a), (b) and Def. 4.1.6.

□

9. We wish to prove that

$$s_0 \models AG[a\boxtimes ! \wedge a.Reply](A[a.pledge_{\{true\}} W_{\{a\checkmark ?\vee a\text{✗}?\vee a.useBy\leq time\}} true])$$

Proof Let σ be a path from s_0, $0 \leq i$ and $\sigma(i, i+1)$ be such that $a\boxtimes ? \in \alpha_{i+1}$ and $a.Reply^{\Theta^{\sigma(i)\rightarrow\sigma(i+1)}} = true$. In this case $\sigma(i, i+1) \models a\boxtimes ? \wedge a.Reply$.

(a) Let $sigma''$ be a path from $\sigma(i+1)$ that contains a transition $r' = s_1 \rightarrow s_1'$ such that $a\checkmark \in EXC^{r'}$ or $a\text{✗} \in EXC^{r'}$ or $a.useBy \leq TIME^{s_1'}$ and there is no $s_2 < s_1'$ where $a.useBy \leq TIME^{s_2}$ (i.e. s_1' is the first state in which the deadline has elapsed). In this case there is $0 < j$ such that $\sigma(j-1, j) \models a\checkmark ? \vee a\text{✗}? \vee a.useBy \leq time$.

Let $i' < j$. Since according to lemma A.1.1 there is no other transition $r'' < r$ such that $a\checkmark \in EXC^{r''}$ or $a\text{✗} \in EXC^{r''}$, we

conclude from Def. 3.5.2 that $a.pledge \in PLG^{\sigma''(i')}$. That is, for every $0 < i' < j$, $\sigma'' \models a.pledge$. We conclude from 4.1.6 that

$$\sigma'' \models a.pledge_{\{true\}}U_{\{a\checkmark?\vee a\text{✗}?\vee a.useBy\leq time\}}true]$$

(b) Let $\sigma' = (\sigma(i, i+1))\sigma''$ be such that there is NO transition $r' = s_1 \rightarrow s_1' \in \sigma''$ where $a\checkmark \in EXC^{r'}$ or $a\text{✗} \in EXC^{r'}$ or $a.useBy \leq TIME^{s_1'}$. From Def. 3.5.2 and Def. 3.4.4 (time moves forward) we conclude that for every state $\sigma''(i'')$ where $i \leq i''$, $a.pledge \in PLG^{\sigma''(i'')}$, i.e. for every $i \leq i''$, $\sigma'' \models a.pledge$.

(c) We conclude from (a) and (b) that

$$\sigma'' \models a.pledge_{\{true\}}W_{\{a\checkmark?\vee a\text{✗}?\vee a.useBy\leq time\}}true$$

From Def. 4.1.6 we conclude that

$$\sigma(i+1) \models A[a.pledge_{\{true\}}W_{\{a\checkmark?\vee a\text{✗}?\vee a.useBy\leq time\}}true]$$

and from Def. 4.1.7 we conclude that

$$\sigma(i) \models [a\text{✉}! \wedge a.Reply]A[a.pledge_{\{true\}}W_{\{a\checkmark?\vee a\text{✗}?\vee a.useBy\leq time\}}true]$$

The result follows immediately. □

10. We wish to prove that

$$s_0 \models A[true_{\{\neg a\text{✞}?\wedge\neg a\text{✞}¿\}}W_{\{a\checkmark?\vee a\checkmark¿\}}true]$$

Proof

(a) Let σ be a path from s_0 that contains a transition $s \xrightarrow{\alpha} s'$ where $a\checkmark? \in \alpha$ or $a\checkmark¿ \in \alpha$. In this case, there is $j > 0$, where $\sigma(j-1) = s$, $a\checkmark? \in \alpha$ or $a\checkmark¿ \in \alpha$ and $\sigma(j-1) = s'$. This means that $\sigma(j-1, j) \models a\checkmark? \vee a\checkmark¿$.

Let $0 < i < j$. If $\sigma(i-1,i) \models a\Updownarrow?$ or $\sigma(i-1,i) \models a\Updownarrow¿$ then by clause 10. in Def. 3.5.2 there should be a $i' < i$ such that $\sigma(i'-1,i') \models a\checkmark?$ or $\sigma(i'-1,i') \models a\checkmark¿$. But this contradicts lemma A.1.1 (according to which events can only be processed once). Hence, $\sigma(i-1,i) \not\models a\Updownarrow?$ and $\sigma(i-1,i) \not\models a\Updownarrow¿$, i.e. $\sigma(i-1,i) \models \neg a\Updownarrow? \wedge \neg a\Updownarrow¿$.

Therefore, according to Def. 4.1.6, $\sigma \models true_{\{\neg a\Updownarrow? \wedge \neg a\Updownarrow¿\}} U_{\{a\checkmark? \vee a\checkmark¿\}} true$.

(b) Let σ be a path from s_0 that does NOT contain a transition $s \xrightarrow{\alpha} s'$ where $a\checkmark? \in \alpha$ or $a\checkmark¿ \in \alpha$. Then, according to Def. 3.5.2 σ cannot contain a transition $s_1 \xrightarrow{\alpha_1} s_1'$ with $a\Updownarrow? \in \alpha$ or $a\Updownarrow¿ \in \alpha$. That is, for all $i \geq 0$, $\sigma(i-1,i) \models \neg a\Updownarrow? \wedge \neg a\Updownarrow¿$.

(c) The result follows from (a), (b) and Def. 4.1.6.

□

A.6.2 ⇐

Let us assume that s_0 satisfies every formula in 4.3.5 and $r = s \longrightarrow s'$ is a transition of m.

1. We wish to prove that if $a\boxtimes \in PUB^r$ then there is $r' < r$ such that $a\spadesuit \in EXC^{r'}$.

Proof Let us assume that $a\boxtimes \in PUB^r$ and σ is a path from s_0 that contains r. In this case, there is $i > 0$ such that $\sigma(i-1,i) \models a\boxtimes!$. Since according to 4.3.5, $s_0 \models A[true_{\{\neg a\boxtimes!\}} W_{\{a\spadesuit?\}} true]$ then we conclude that

$$\sigma \models true_{\{\neg a\boxtimes!\}} W_{\{a\spadesuit?\}} true$$

Because there is $i > 0$ such that $\alpha_i \models a\boxtimes!$, then we also conclude from

4.1.6 that

$$\sigma \models true_{\{\neg a\boxtimes !\}}U_{\{a\text{🔔}?\}}true$$

i.e. there exists $j \geq 1$ such that $\sigma(j-1, j) \models a$🔔? and for all i such that $0 < i' < j$, $\sigma(i'-1, i') \models \neg a\boxtimes !$. Since $\sigma(i-1, i) \models a\boxtimes !$, we conclude that $j < i$. That is, there is a transition $r' < r$ such that a🔔 $\in EXC^{r'}$ (as we wanted to prove). □

2. We wish to prove that if a🔔 $\in EXC^r$, then there is $r < r'$ such that $a\boxtimes \in PUB^{r'}$.

Proof Let us assume that a🔔 $\in EXC^r$ and $\sigma = \sigma'(\sigma(i-1) \xrightarrow{\alpha_i} \sigma(i))\sigma''$, where $\sigma_{i-1} = s$ and $\sigma_i = s'$, is a path from s_0. In this case, there is $i > 0$ such that $\sigma(s_{i-1}, s_i) \models a$🔔?. According to 4.3.5

$$s_0 \models AG[a\text{🔔}?](A[true_{\{true\}}U_{\{a\boxtimes !\}}true])$$

We conclude from Def. 4.1.6 that

$$s \models [a\text{🔔}?](A[true_{\{true\}}U_{\{a\boxtimes !\}}true])$$

Since $s \xrightarrow{\alpha} s' \models a$🔔?, we can conclude that

$$s' \models A[true_{\{true\}}U_{\{a\boxtimes !\}}true])$$

and that

$$\sigma'' \models true_{\{true\}}U_{\{a\boxtimes !\}}true]$$

which means that there is $i < j$ such that $\sigma(j-1) \xrightarrow{\alpha_j} \sigma(j) \models a\boxtimes !$. Hence, there is $r' = \sigma(j-1) \longrightarrow \sigma(j)$ such that $r < r'$ and $a\boxtimes \in PUB^{r'}$. □

3. We wish to prove that if $a\checkmark \in EXC^r$ then there is $r' < r$ such that $a\boxtimes \in PUB^{r'}$, $a.reply^{\Theta^{r'}} = true$ and $TIME^{s'} < a.useBy^{\Theta^{r'}}$.

Proof Let us assume that $a\checkmark \in EXC^r$ and σ is a path from s_0 that contains r. In this case, there is $i > 0$ such that $\sigma = \sigma'(\sigma(i-1,i))\sigma''$ and $\sigma(i-1,i) \models a\checkmark?$. Since according to 4.3.5,

$$s_0 \models A[true_{\{\neg a\checkmark?\}} W_{\{a\boxtimes!\wedge a.Reply\}} A[true_{\{\neg a\checkmark?\}} W_{\{a\checkmark?\wedge time<a.useBy\}} true]]$$

then

$$\sigma \models true_{\{\neg a\checkmark?\}} W_{\{a\boxtimes!\wedge a.Reply\}} A[true_{\{\neg a\checkmark?\}} W_{\{a\checkmark?\wedge time<a.useBy\}} true]$$

Because there is $i > 0$ such that $\sigma(i-1,i) \models a\checkmark$, it is not the case that for every $0 < j$, $\sigma(j-1,j) \models \neg a\checkmark?$. Therefore we conclude from 4.1.6 that

$$true_{\{\neg a\checkmark?\}} U_{\{a\boxtimes!\wedge a.Reply\}} A[true_{\{\neg a\checkmark?\}} W_{\{a\checkmark?\wedge time<a.useBy\}} true]$$

That is, there is $0 < i''$ such that $\sigma(i') \models a\boxtimes!\wedge a.Reply$. Since for every $j' < i'$, $\sigma(j'-1,j') \models \neg a\checkmark?$ and since $\sigma(i-1,i) \models a\checkmark?$, we conclude that $i' < i$. That is, there is a transition $r' < r$ such that $a\boxtimes \in PUB^{r'}$ and $a.reply^{\Theta^{r'}} = true$.

We also conclude from 4.1.6 that

$$\sigma(i) \models A[true_{\{\neg a\checkmark?\}} W_{\{a\checkmark?\wedge time<a.useBy\}} true]$$

and that

$$\sigma'' \models true_{\{\neg a\checkmark?\}} W_{\{a\checkmark?\wedge time<a.useBy\}} true$$

Since $i' < i$, it is the case that there is $0 < i''$ such that $\sigma''(i''-1,i'') \models a\checkmark? \wedge time < a.useBy$. From lemma A.1.1 we conclude that $\sigma(i-1,i) = \sigma''(i''-1,i'')$, Therefore, $\sigma(i-1,i) \models a\checkmark? \wedge time < a.useBy$, which according to Def. 4.1.6 means that $TIME^{s'} < a.useBy^{\Theta^{r'}}$.

□

4. We wish to prove that if $a\checkmark \in EXC^r$ then there is no $r' < r$ such that $a\text{✗} \in EXC^{r'}$.

   Proof Let us assume that $a\checkmark \in EXC^r$ and σ is a path from s_0 that contains r. In this case, there is $i > 0$ such that $\sigma(i-1, i) \models a\checkmark?$. Let $\sigma = \sigma'(s_1 \xrightarrow{\alpha'} s_1')\sigma''$ where $r' = s_1 \rightarrow s_1' < r$

   Let us suppose that $a\text{✗} \in EXC^{r'}$. In this case $s_1 \xrightarrow{\alpha'} s_1' \models a\text{✗}?$. Since according to 4.3.5

   $$s_0 \models AG[a\text{✗}?](A[true_{\{\neg a\checkmark\}}Wfalse])$$

   we conclude from Def. 4.1.7 that

   $$s_1 \models [a\text{✗}?](A[true_{\{\neg a\checkmark?\}}Wfalse])$$

   Since $s_1 \xrightarrow{\alpha'} s_1' \models a\text{✗}?$, we can conclude that

   $$s_1' \models A[true_{\{\neg a\checkmark?\}}Wfalse]$$

   and that

   $$\sigma'' \models true_{\{\neg a\checkmark?\}}Wfalse$$

   which means that for every $i < j$, $\sigma(j-1, j) \models \neg a\checkmark?$, i.e. $a\checkmark? \notin \alpha_j$. But this contradicts the fact that $r' < r$ and $r \models a\checkmark?$. Hence, $a\text{✗} \notin EXC^{r'}$. That is, there is no transition $r' < r$ in σ such that $a\text{✗} \in EXC^{r'}$.

   □

5. We wish to prove that if $a\text{✗} \in EXC^r$ then there is $r' < r$ such that $a\text{✉} \in PUB^{r'}$, $a.reply^{\Theta^{r'}} = true$ and $TIME^{s'} < a.useBy^{\Theta^{r'}}$

   Proof Let us assume that $a\text{✗} \in EXC^r$ and σ is a path from s_0 that contains r. In this case, there is $i > 0$ such that $\sigma = \sigma'(\sigma(i-1, i))\sigma''$

and $\sigma(i-1,i) \models a\text{✗}?$. Since according to 4.3.5,

$$s_0 \models A[true_{\{\neg a\text{✗}?\}} W_{\{a\boxtimes !\wedge a.Reply\}} A[true_{\{\neg a\text{✗}?\}} W_{\{a\text{✗}?\wedge time<a.useBy\}} true]]$$

then

$$\sigma \models true_{\{\neg a\text{✗}?\}} W_{\{a\boxtimes !\wedge a.Reply\}} A[true_{\{\neg a\text{✗}?\}} W_{\{a\text{✗}?\wedge time<a.useBy\}} true]$$

Because there is $i>0$ such that $\sigma(i-1,i) \models a\text{✗}$, it is not the case that for every $0<j$, $\sigma(j-1,j) \models \neg a\text{✗}?$. Therefore we conclude from 4.1.6 that

$$true_{\{\neg a\text{✗}?\}} U_{\{a\boxtimes !\wedge a.Reply\}} A[true_{\{\neg a\text{✗}?\}} W_{\{a\text{✗}?\wedge time<a.useBy\}} true]$$

That is, there is $0<i''$ such that $\sigma(i''-1,i'') \models a\boxtimes ! \wedge a.Reply$. Since for every $j'<i'$, $\sigma(j'-1,j') \models \neg a\text{✗}?$ and since $\sigma(i-1,i) \models a\text{✗}?$, we conclude that $i'<i$. That is, there is a transition $r'<r$ such that $a\boxtimes \in PUB^{r'}$ and $a.reply^{\Theta^{r'}} = true$.

We also conclude from 4.1.6 that

$$\sigma(i) \models A[true_{\{\neg a\text{✗}?\}} W_{\{a\text{✗}?\wedge time<a.useBy\}} true]$$

and that

$$\sigma'' \models true_{\{\neg a\text{✗}?\}} W_{\{a\text{✗}?\wedge time<a.useBy\}} true$$

Since $i'<i$, it is the case that there is $0<i''$ such that $\sigma''(i''-1,i'') \models a\text{✗}?\wedge time<a.useBy$. From lemma A.1.1 we conclude that $\sigma(i-1,i)=\sigma''(i''-1,i'')$, Therefore, $\sigma(i-1,i) \models a\text{✗}?\wedge time<a.useBy$, which according to Def. 4.1.6 means that $TIME^{s'} < a.useBy^{\Theta^{r'}}$.

□

6. We wish to prove that if $a\text{✗} \in EXC^r$ then there is no $r'<r$ such that $a\checkmark \in EXC^{r'}$

Proof Let us assume that $a\text{✗} \in EXC^r$ and σ is a path from s_0 that contains r. In this case, there is $i > 0$ such that $\sigma(i-1, i) \models a\text{✗}?$. Let $\sigma = \sigma'(s_1 \xrightarrow{\alpha'} s'_1)\sigma''$ where $r' = s_1 \rightarrow s'_1 < r$.

Let us suppose that $a\checkmark \in EXC^{r'}$. In this case $s_1 \xrightarrow{\alpha'} s'_1 \models a\checkmark?$. Since according to 4.3.5

$$s_0 \models AG[a\checkmark?](A[true_{\{\neg a\text{✗}\}} W false])$$

we conclude from Def. 4.1.7 that

$$s_1 \models [a\checkmark?](A[true_{\{\neg a\text{✗}?\}} W false])$$

Since $s_1 \xrightarrow{\alpha'} s'_1 \models a\checkmark?$, we can conclude that

$$s'_1 \models A[true_{\{\neg a\text{✗}?\}} W false]$$

and that

$$\sigma'' \models true_{\{\neg a\text{✗}?\}} W false$$

which means that for every $i < j$, $\sigma(j) \models \neg a\text{✗}?$, i.e. $a\text{✗}? \notin \alpha_j$. But this contradicts the fact that $r' < r$ and $r \models a\text{✗}?$. Hence, $a\checkmark \notin EXC^{r'}$. That is, there is no transition $r' < r$ in σ such that $a\checkmark \in EXC^{r'}$.

□

7. We wish to prove that if $a\text{✉} \in PUB^r$ and $a.Reply^{\theta^r}$, then there is no transition $r' = s_1 \xrightarrow{\alpha} s'_1$ with $r < r'$ such that:

   - $a\checkmark \in DSC^{r'}$
   - $TIME^{n_1} < a.useBy^{\Theta^r}$
   - there is no transition $r < r'' < r'$ such that $a\text{✗} \in EXC^{r''}$

   Proof Let us assume that $a\text{✉} \in PUB^r$, $a.Reply^{\theta^r}$ and σ is a path from s_0 that contains r. In this case, there is $i > 0$ such that $\sigma =$

$\sigma'(\sigma(i-1,i))\sigma''$ and $\sigma(i-1,i) \models a\boxtimes! \wedge a.Reply$. Since according to 4.3.5,

$$s_0 \models A[true_{\{\neg(a\boxtimes!\wedge a.Reply)\}} W_{\{a\boxtimes!\wedge a.Reply\}} \neg E[true_{\{\neg a\text{✗}?\}} U_{\{a\checkmark¿\wedge time<a.useBy\}} true]]$$

then

$$\sigma \models true_{\{\neg(a\boxtimes!\wedge a.Reply)\}} W_{\{a\boxtimes!\wedge a.Reply\}} \neg E[true_{\{\neg a\text{✗}?\}} U_{\{a\checkmark¿\wedge time<a.useBy\}} true$$

Because there is $i > 0$ such that $\sigma(i-1,i) \models a\boxtimes! \wedge a.Reply = true$, we also conclude from 4.1.6 that

$$\sigma \models true_{\{\neg(a\boxtimes!\wedge a.Reply)\}} U_{\{a\boxtimes!\wedge a.Reply\}} \neg E[true_{\{\neg a\text{✗}?\}} U_{\{a\checkmark¿\wedge time<a.useBy\}} true]$$

i.e.

$$\sigma(i) = s' \models \neg E[true_{\{\neg a\text{✗}?\}} U_{\{a\checkmark¿\wedge time<a.useBy\}} true]$$

We conclude from 4.1.6 that

$$\sigma'' \not\models true_{\{\neg a\text{✗}?\}} U_{\{a\checkmark¿\wedge time<a.useBy\}} true$$

which means that there is no $i < j$ such that $\sigma(j-1,j) \models a\checkmark¿ \wedge time < a.useBy$ and such that for all $i < j' < j$, $\sigma(j'-1,j') \models \neg a\text{✗}?$. That is, there is no transition $r' = s_1 \xrightarrow{\alpha} s_1'$ in σ with $r < r'$ such that:

- $a\checkmark \in DSC^{r'}$
- $TIME^{n_1} < a.useBy^{\Theta^r}$
- there is no transition $r < r'' < r'$ in σ such that $a\text{✗} \in EXC^{r''}$

□

8. We wish to prove that if $a\boxtimes \in PUB^r$ and $a.reply^{\Theta^r} = true$, then there is no $r' = n_1 \longrightarrow n_2$ with $r < r'$ such that:

   - $a\text{✗} \in DSC^{r'}$

- $TIME^{n_1} < a.useBy^{\Theta^r}$
- there is no transition $r < r'' < r'$ such that $a\checkmark \in EXC^{r''}$

Proof Let us assume that $a\boxtimes \in PUB^r$, $a.Reply^{\theta^r}$ and σ is a path from s_0 that contains r. In this case, there is $i > 0$ such that $\sigma = \sigma'(\sigma(i-1,i))\sigma''$ and $\sigma(i-1,i) \models a\boxtimes! \wedge a.Reply$. Since according to 4.3.5,

$$s_0 \models A[true_{\{\neg(a\boxtimes!\wedge a.Reply)\}} W_{\{a\boxtimes!\wedge a.Reply\}} \neg E[true_{\{\neg a\checkmark?\}} U_{\{a\text{✗¿}\wedge time<a.useBy\}} true]]$$

then

$$\sigma \models true_{\{\neg(a\boxtimes!\wedge a.Reply)\}} W_{\{a\boxtimes!\wedge a.Reply\}} \neg E[true_{\{\neg a\checkmark?\}} U_{\{a\text{✗¿}\wedge time<a.useBy\}} true$$

Because there is $i > 0$ such that $\sigma(i-1,i) \models a\boxtimes! \wedge a.Reply = true$, we also conclude from 4.1.6 that

$$\sigma \models true_{\{\neg(a\boxtimes!\wedge a.Reply)\}} U_{\{a\boxtimes!\wedge a.Reply\}} \neg E[true_{\{\neg a\checkmark?\}} U_{\{a\text{✗¿}\wedge time<a.useBy\}} true]$$

i.e.

$$\sigma(i) = s' \models \neg E[true_{\{\neg a\checkmark?\}} U_{\{a\text{✗¿}\wedge time<a.useBy\}} true]$$

We conclude from 4.1.6 that

$$\sigma'' \not\models true_{\{\neg a\checkmark?\}} U_{\{a\text{✗¿}\wedge time<a.useBy\}} true$$

which means that there is no $i < j$ such that $\sigma(j-1,j) \models a\text{✗¿} \wedge time < a.useBy$ and such that for all $i < j' < j$, $\sigma(j'-1,j') \models \neg a\checkmark?$. That is, there is no transition $r' = s_1 \xrightarrow{\alpha} s_1'$ in σ with $r < r'$ such that:

- $a\text{✗} \in DSC^{r'}$
- $TIME^{n_1} < a.useBy^{\Theta^r}$
- there is no transition $r < r'' < r'$ in σ such that $a\checkmark \in EXC^{r''}$

□

9. We wish to prove that if $a.pledge \in PLG^{s'}$ if the following conditions hold:

   - there is a transition $r' \in R$ such that $r' \leq s'$ and $a\boxtimes \in PUB^{r'}$ and $a.reply^{\Theta^{r'}} = true$
   - there is no $r'' < n'$ with $a\checkmark \in EXC^{r''}$ or $a\text{✗} \in EXC^{r''}$
   - $TIME^{n'} < a.useBy^{\Theta^{r'}}$

   Proof

   Let us assume that:

   - there is a transition $r' = s_1 \rightarrow s'_1 \in R$ such that $r' \leq s'$ and $a\boxtimes \in PUB^{r'}$ and $a.reply^{\Theta^{r'}} = true$
   - there is no $r'' < n'$ with $a\checkmark \in EXC^{r''}$ or $a\text{✗} \in EXC^{r''}$
   - $TIME^{n'} < a.useBy^{\Theta^{r'}}$

   Let $\sigma = \sigma'(\sigma(i-1,i))\sigma''$ be such that $\sigma(i-1) = s_1$ and $\sigma(i) = s'_1$. In this case $\sigma(i-1,i) \models a\boxtimes! \wedge a.Reply$. According to 4.3.5

   $$s_0 \models AG[a\boxtimes! \wedge a.Reply](A[a.pledge_{\{true\}} W_{\{a\checkmark ? \vee a\text{✗} ? \vee a.useBy \leq time\}} true])$$

   We conclude from Def. 4.1.7 that

   $$s'_1 \models A[a.pledge_{\{true\}} W_{\{a\checkmark ? \vee a\text{✗} ? \vee a.useBy \leq time\}} true]$$

   and that

   $$\sigma'' \models a.pledge_{\{true\}} W_{\{a\checkmark ? \vee a\text{✗} ? \vee a.useBy \leq time\}} true$$

   We conclude that either:

   (a) $\sigma'' \models a.pledge_{\{true\}} U_{\{a\checkmark ? \vee a\text{✗} ? \vee a.useBy \leq time\}} true$;

   In this case there is $0 < j$ such that $\sigma'' \models a\checkmark ? \vee a\text{✗} ? \vee a.useBy \leq time$ and for every $0 < i' < j$, $\sigma''(j) \models a.pledge$. Since $r' < r$,

we conclude that $s' = \sigma''(i')$ for some $0 < i''$. And since there is no $r'' < n'$ with $a\checkmark \in EXC^{r''}$ or $a\text{✗} \in EXC^{r''}$, we conclude that $i'' < j$. Therefore, $\sigma''(i'') = s' \models a.pledge$, i.e. $a.pledge \in PLG^{s'}$.

(b) or, for every $0 < j$, $\sigma''(j) \models a.pledge$. In this case, since $r' < r$, we conclude that r is in σ''. Therefore, $s' \models a.pledge$, i.e. $a.pledge \in PLG^{s'}$.

The result follows immediately from (a) and (b).

□

10. We wish to prove that if $a\Updownarrow \in PRC^r$, then there is $r' \in R$ such that $r' < r$ and $a\checkmark \in PRC^{r'}$.

Proof Let us assume that $a\Updownarrow \in PRC^r$ and σ is a path from s_0 that contains r. Since $PRC = EXC \uplus DSC$, in this case, there is $i > 0$ such that $a\Updownarrow? \in \alpha_i$ or $a\Updownarrow¿ \in \alpha_i$, i.e. $\alpha_i \models a\Updownarrow? \vee a\Updownarrow¿$. Since according to 4.3.5,

$$s_0 \models A[true_{\{\neg a\Updownarrow? \wedge \neg a\Updownarrow¿\}} W_{\{a\checkmark? \vee a\checkmark¿\}} true]$$

we conclude that

$$\sigma \models true_{\{\neg a\Updownarrow? \wedge \neg a\Updownarrow¿\}} W_{\{a\checkmark? \vee a\checkmark¿\}} true$$

Because there is $i > 0$ such that $\alpha_i \models a\Updownarrow? \vee a\Updownarrow¿$, then we also conclude from 4.1.6 that

$$\sigma \models true_{\{\neg a\Updownarrow? \wedge \neg a\Updownarrow¿\}} U_{\{a\checkmark? \vee a\checkmark¿\}} true$$

i.e. there exists $j \geq 1$ such that $\alpha_j \models a\Updownarrow? \vee a\Updownarrow¿$ and for all i such that $0 < i' < j$, $\alpha_i \models \neg a\Updownarrow? \wedge \neg a\Updownarrow¿$. Since $\alpha_i \models a\Updownarrow? \vee a\Updownarrow¿$, we conclude that $j < i$. That is, there is a transition $r' < r$ such that $a\Updownarrow \in EXC^{r'}$ or $a\Updownarrow \in DSC^{r'}$, which means that $a\Updownarrow \in PRC^{r'}$ (as we wanted to prove).
□

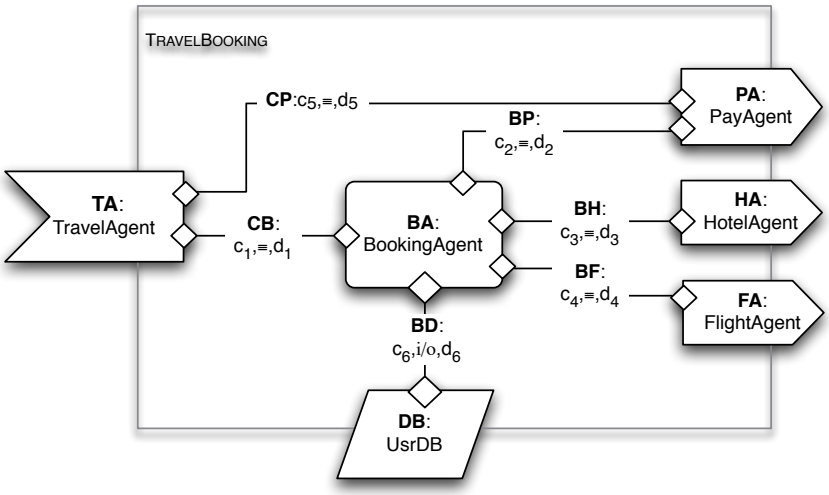
Appendix B

The *TravelBooking* case study

In this appendix, we define the *TravelBooking* service module. We give a graphical overview of the structure of this service module for convenience before specifying it using the SENSORIA Reference Modelling Language – SRML.

The persistent component *DB* provides three synchronous interactions *log*, *getData* and *getCard* with type *rpl*. These interactions are modelled as functions that can be used in the orchestration of the co-party of *DB* (i.e. *BA*).

The interaction protocols involved in this example are "standard". All connectors (except for *BD*) implement a straight (asynchronous) protocol in the sense that events issued by one party are forwarded directly to the co-party without any changes on the parameters. This is denoted by ≡, which actually stands for a family of interaction protocols. For convenience, we only define the interaction protocol that is used by wire *BH*. In the case of BD, we specify a straight (synchronous) input/output protocol, denoted by *i/o*, again in the sense that no operations on data are require.



TRAVELBOOKING consists of:

- TR – the provides-interface of the module, of type *TravelAgent*;
- FA – a requires-interface (for a flight-booking service), of type *FlightAgent*;
- PA – a requires-interface (for a payment service), of type *PayAgent*;
- HA – a requires-interface (for a hotel-booking service), of type *HotelAgent*;
- BA – an interface for component that coordinates the business process, of type *BookingAgent*;

- DB – a uses-interface for the persistent component (of the bottom layer) that stores user data, of type *UsrDB*;
- *CB*, *CP*, *BF*, *BH*, *BP*, *BD* – wire-interfaces typed by connectors that establish the required interconnections.

MODULE TravelBooking **is**

DATATYPE

sorts: usrname, password, usrdata, boolean, fcode, hcode, pCode, airport, date, payData, accountn, moneyvalue, serviceId, nat

COMPONENTS

BA: BookingAgent

PROVIDES

CR: Customer

REQUIRES

FA: FlightAgent
PA: PayAgent
HA: HotelAgent

USES

DB: UserDB

WIRES

BA BookingAgent	c_4	**BF**	d_4	**FA** FlightAgent
s&r bookFlight	S		R	**r&s** lockFlight
🔔 from	i_1		i_1	🔔 from
to	i_2		i_2	to
out	i_3		i_3	out
in	i_4	〓	i_4	in
traveller	i_5		i_5	traveller
✉ fconf	o_1		o_1	✉ fconf
amount	o_2		o_2	amount
beneficiary	o_3		o_3	beneficiary
payService	o_4		o_4	payService
snd payAck	S		R	**rcv** payAck
🔔 proof	i_1	〓	i_1	🔔 proof
status	i_2		i_2	status

rcv ackRefundRcv 🔔 amount	R i_1	≡	S i_1	**snd** payRefund 🔔 amount

BA BookingAgent	c_3	**BH**	d_3	**HA** HotelAgent
s&r bookHotel	S		R	**r&s** lockHotel
🔔 checkin	i_1		i_1	🔔 checkin
checkout	i_2	Straight	i_2	checkout
traveller	i_3	(date,date,	i_3	name
✉ hconf	o_1	usrdata,hcode)	o_1	✉ hconf

BA BookingAgent	c_6	**BD**	d_6	**DB** UserDB
ask log	A	**i/o**	R	**rpl** log
ask getData	A	**i/o**	R	**rpl** getData
ask getCard	A	**i/o**	R	**rpl** getCard

BA BookingAgent	c_2	**BP**	d_2	**PA** PayAgent
s&r payment	S		R	**r&s** payment
🔔 amount	i_1		i_1	🔔 amount
beneficiary	i_2	≡	i_2	beneficiary
originator	i_3		i_3	originator
cardNo	i_4		i_4	cardNo
✉ proof	o_1		o_1	✉ proof

END MODULE

SPECIFICATIONS

LAYER PROTOCOL UsrDB **is**

INTERACTIONS

rpl log(usrname,password):bool

rpl getData(usrname):usrdata

rpl getCard(usrname):paydata

BEHAVIOUR

END LAYER PROTOCOL

```
BUSINESS ROLE BookingAgent is

  INTERACTIONS
    r&s login
      🔔 usr:usrname,
         pwd:password
      ✉ fconf:fcode,
         hconf:hcode,
         amount:moneyvalue
    r&s bookTrip
      🔔 from,to:airport,
         out,in:date
      ✉ fconf:fcode,
         hconf:hcode,
         amount:moneyvalue
    ask log(username,password):bool
    ask getData(username):usrdata
    ask getCard(username):paydata
    s&r bookFlight
      🔔 from,to:airport,
         out,in:date,
         traveller:usrdata
      ✉ fconf:fcode
         amount:moneyvalue
         beneficiary:accountn
         payService:serviceId
    s&r payment
      🔔 amount:moneyvalue
         beneficiary:accountn
         originator:usrdata
         cardNo:paydata
      ✉ proof:pcode
    s&r bookHotel
      🔔 checkin:date, checkout:date,
         traveller:usrdata
      ✉ hconf:hcode
    snd payAck
      🔔 proof:pCode,
         status:bool
    rcv ackRefundRcv
      🔔 amount:moneyvalue
    snd ackRefundSnd
      🔔 amount:moneyvalue

  ORCHESTRATION
    local
      s:[START, LOGGED, QUERIED, FLIGHT_OK, HOTEL_OK,
         CONFIRMED, END_PAYED, END_UNBOOKED,
         COMPENSATING, END_COMPENSATED],
      login:boolean,
      traveller:usrdata,
      travcard:paydata

    transition Login
      triggeredBy login🔔
      guardedBy s=START
      effects login'=log(login🔔.usrname,login🔔.pwd)
         ∧ login' ⇒ (s'=LOGGED
              ∧ traveller'=getData(login.usrname)
```

```
            ∧ travcard'=getCard(login.usrname))
        ∧ ¬login' ⇒ s'=END_UNBOOKED
        ∧ login✉ ∧ login.Reply=login'
transition Request
  triggeredBy bookTrip🔔
  guardedBy s=LOGGED
  effects s'=QUERIED
        ∧ bookTrip🔔.out>today ⇒ bookFlight🔔
        ∧ bookFlight🔔.from=bookTrip🔔.from
        ∧ bookFlight🔔.to=bookTrip🔔.to
        ∧ bookFlight🔔.out=bookTrip🔔.out
        ∧ bookFlight🔔.in=bookTrip🔔.in
        ∧ bookFlight🔔.traveller=traveller
        ∧ bookTrip🔔.out≤today ⇒ bookTrip✉
        ∧ bookTrip✉.Reply=False
transition FlightAnswer
  triggeredBy bookFlight✉
  guardedBy s=QUERIED
  effects bookFlight✉.Reply ⇒ s'=FLIGHT_OK
        ∧ ¬bookFlight✉.Reply ⇒ s'=END_UNBOOKED
        ∧ bookFlight✉.Reply ⇒ bookHotel🔔
        ∧ bookHotel🔔.checkin=bookTrip🔔.in
        ∧ bookHotel🔔.checkout=bookTrip🔔.out
        ∧ bookHotel🔔.traveller=traveller
        ∧ ¬bookFlight✉.Reply ⇒ bookTrip✉
        ∧ bookTrip✉.Reply=False
transition HotelAnswer
  triggeredBy bookHotel✉
  guardedBy s=FLIGHT_OK
  effects bookHotel✉.Reply ⇒ s'=HOTEL_OK
        ∧ ¬bookHotel✉.Reply ⇒ s'=END_UNBOOKED
        ∧ bookHotel✉.Reply ⇒ bookTrip✉
        ∧ bookTrip✉.fconf=bookFlight✉.fconf
        ∧ bookTrip✉.amount=bookFlight✉.amount
        ∧ bookTrip✉.hconf=bookHotel✉.hconf
        ∧ ¬bookHotel✉.Reply ⇒ bookFlight✖
        ∧ bookTrip✉ ∧ bookTrip✉.Reply=False
transition TripCommit
  triggeredBy bookTrip✓
  guardedBy s=HOTEL_OK
  effects s'=CONFIRMED
        ∧ bookFlight✓ ∧ bookHotel✓ ∧ payment🔔
        ∧ payment🔔.amount=bookFlight✉.amount
        ∧ payment🔔.beneficiary=
            bookFlight✉.beneficiary
        ∧ payment🔔.originator=traveller
        ∧ payment🔔.cardNo=travcard
transition PaymentAnswer
  triggeredBy payment✉
  guardedBy s=CONFIRMED
  effects payment✉.Reply ⇒ s'=END_PAYED
        ∧ ¬payment✉.Reply ⇒ s'=END_UNBOOKED
        ∧ payAck🔔 ∧ payAck🔔.proof=payment✉.proof
        ∧ payAck🔔.status=payment✉.Reply
transition TripCancel
  triggeredBy bookTrip✖
  guardedBy s=HOTEL_OK
  effects s'=END_UNBOOKED
```

```
        ∧ bookFlight✖ ∧ bookHotel✖
transition TripCompensate
  triggeredBy bookTrip⇧
  guardedBy s=END_PAYED ∧ today<out
  effects s'=COMPENSATING
        ∧ bookFlight⇧ ∧ bookHotel⇧
transition TripRefund
  triggeredBy ackRefundRcv🔔?
  guardedBy s=COMPENSATING
  effects s'=END_COMPENSATED
        ackRefundSnd🔔
        ∧ ackRefundSnd🔔.amount=ackRefundRcv🔔.amount
```

END BUSINESS ROLE

BUSINESS PROTOCOL FlightAgent **is**

```
INTERACTIONS
  r&s lockFlight
     🔔 from,to:airport,
        out,in:date,
        traveller:usrdata
     ✉ fconf:fcode
        amount:moneyvalue,
        beneficiary:accountn,
        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⇧?
  lockFlight⇧? ensures payRefund🔔!
```

END BUSINESS PROTOCOL

BUSINESS PROTOCOL HotelAgent **is**

```
INTERACTIONS
  r&s lockHotel
     🔔 checkin,checkout:date
        name:usrdata
     ✉ hconf:hcode
BEHAVIOUR
  initiallyEnabled lockHotel🔔?
  lockHotel✓? enables lockHotel⇧?
```

END BUSINESS PROTOCOL

BUSINESS PROTOCOL PayAgent **is**

INTERACTIONS

r&s payment
🔔 amount:moneyvalue
beneficiary:accountn
originator:usrdata
cardNo:paydata
✉ proof:pcode

snd payNotify
🔔 status:bool

BEHAVIOUR

initiallyEnabled payment🔔?
payment✉! **ensures** payNotify🔔!

END BUSINESS PROTOCOL

BUSINESS PROTOCOL TravelAgent **is**

INTERACTIONS

r&s login
🔔 usr:username, pwd:password

r&s bookTrip
🔔 from,to:airport,
out,in:date
✉ fconf:fcode,
hconf:hcode,
amount:moneyvalue

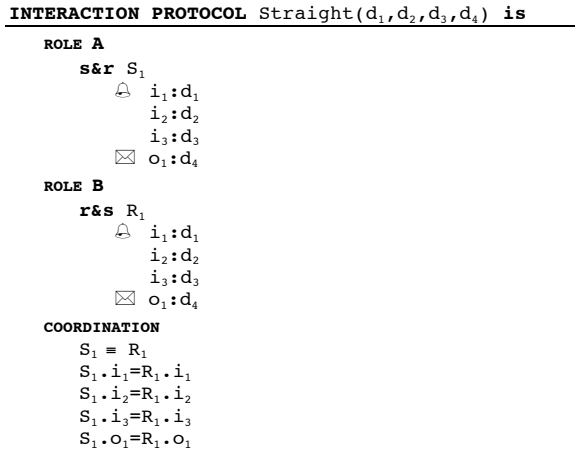
snd payNotify
🔔 status:boolean

snd refund
🔔 amount:moneyvalue

BEHAVIOUR

initiallyEnabled login🔔?
login✉! ∧ login✉.Reply **enables** bookTrip🔔?
bookTrip✓? **ensures** payNotify🔔!
payNotify🔔! ∧ payNotify🔔.status **enables** bookTrip✞?
bookTrip✞? **ensures** refund🔔!

END BUSINESS PROTOCOL


INTERACTION PROTOCOL Straight(d1,d2,d3,d4) is
ROLE A
s&r S1
i1:d1
i2:d2
i3:d3
o1:d4
ROLE B
r&s R1
i1:d1
i2:d2
i3:d3
o1:d4
COORDINATION
S1 ≡ R1
S1.i1=R1.i1
S1.i2=R1.i2
S1.i3=R1.i3
S1.o1=R1.o1

Appendix C

Encoding of *TravelBooking* with UMC

```
-----------------------------------
----------------WIRES --------------
-----------------------------------

Class  payAck_Wire is
   Signals:
      payAck_bell(proof:Token,status:bool);
   State Top =s1
   Transitions:
      s1 -> s1 { payAck_bell(proof,status) /
                 FAobj.payAck_bell(proof,status) }
end payAck_Wire;

---

Class  refund_Wire is
   Signals:
      payRefund_bell(amount);
   State Top =s1
   Transitions:
      s1 -> s1 { payRefund_bell(amount) /
                 BAobj.ackRefundRcv_bell(amount) }
end refund_Wire;

---

Class  payNotify_Wire is
   Signals:
      payNotify_bell(status:bool);
   State Top =s1
   Transitions:
      s1 -> s1 { payNotify_bell(status) /
                 CRobj.payNotify_bell(status) }
end payNotify_Wire;

---

Class  ackRefundSnd_Wire is
   Signals:
      refund_bell(amount);
   ackRefundSnd_bell(amount)
   State Top =s1
```

```
   Transitions:
      s1 -> s1 { ackRefundSnd_bell(amount)   /
                 CRobj.refund_bell(amount)}
end ackRefundSnd_Wire;

---

Class Hotel_Wire is
   Signals:
      bookHotel_bell(checkin:date,checkout:date,
                     traveller:Token);
      lockHotel_reply(Reply:bool,hconf:Token);
      bookHotel_commit;
      bookHotel_cancel;
      bookHotel_revoke;
   State Top =s1
   Transitions:
      s1 -> s1 { bookHotel_bell(checkin,checkout,traveller) /
                 HAobj.lockHotel_bell(checkin,checkout,
                 traveller)}
      s1 -> s1 { lockHotel_reply(Reply,hconf) /
                 BAobj.bookHotel_reply(Reply,hconf) }
      s1 -> s1 { bookHotel_cancel /
                 HAobj.lockHotel_cancel }
      s1 -> s1 { bookHotel_commit /
                 HAobj.lockHotel_commit }
      s1 -> s1 { bookHotel_revoke /
                 HAobj.lockHotel_revoke }
end Hotel_Wire;

---

Class Flight_Wire is
   Signals:
      bookFlight_bell(from:Token,to:Token,out:date,in:date,
                      traveller:Token);
      lockFlight_reply(Reply:bool,fconf:Token,amount,
                       beneficiary:Token,payService:Token);
      bookFlight_cancel;
      bookFlight_commit;
      bookFlight_revoke;
   State Top =s1
   Transitions:
```

```
      s1 -> s1 { bookFlight_bell(from,to,out,in,traveller)/
                 FAobj.lockFlight_bell(from,to,out,in,
                 traveller)}
      s1 -> s1 { lockFlight_reply(Reply,fconf,amount,
                 beneficiary,payService) /
                 BAobj.bookFlight_reply(Reply,fconf,amount,
                 beneficiary,payService) }
      s1 -> s1 { bookFlight_cancel /
                 FAobj.lockFlight_cancel }
      s1 -> s1 { bookFlight_commit /
                 FAobj.lockFlight_commit }
      s1 -> s1 { bookFlight_revoke /
                 FAobj.lockFlight_revoke }
end Flight_Wire;

---

Class  payment_Wire is
   Signals:
      payment_bell(amount,beneficiary:Token,originator:Token,
                   cardNo:Token);
      payment_reply(Reply:bool,proof:Token);
      payment_cancel;
      payment_commit;
      payment_revoke;
   State Top =s1
   Transitions:
      s1 -> s1 { payment_bell(amount,beneficiary,originator,
                 cardNo)/
                 PAobj.payment_bell(amount,beneficiary,
                 originator,cardNo) }
      s1 -> s1 { payment_reply(Reply,proof) /
                 BAobj.payment_reply(Reply,proof) }
      s1 -> s1 { payment_cancel / PAobj.payment_cancel }
      s1 -> s1 { payment_commit / PAobj.payment_commit }
      s1 -> s1 { payment_revoke / PAobj.payment_revoke }
end payment_Wire;

Class  login_Wire is
   Signals:
      login_bell(usr: Token,pwd:Token);
      login_reply(Reply:bool);
      login_cancel;
```

```
      login_commit;
      login_revoke;
   State Top =s1
   Transitions:
      s1 -> s1 { login_bell (usr,pwd) /
                 BAobj.login_bell(usr,pwd)}
      s1 -> s1 { login_reply(Reply) /
                 CRobj.login_reply(Reply) }
      s1 -> s1 { login_cancel / BAobj.login_cancel }
      s1 -> s1 { login_commit / BAobj.login_commit }
      s1 -> s1 { login_revoke / BAobj.login_revoke }
end login_Wire;

---

Class DB_Wire is
   Operations:
      log(usrname:Token,password:Token): bool;
      getData(usrname: Token): Token;
      getCard(usrname: Token): Token;
   Vars:
      Priority :=2;
   State Top =s1
   Transitions:
      s1 -> s1 { log (usrname,password) /
                 res: bool;
                 res := DBobj.log(usrname,password);
                 return res }
      s1 -> s1 { getData(usrname) /
                 res: usrdata;
                 res := DBobj.getData(usrname);
                 return res }
      s1 -> s1 { getCard(usrname) /
                 res: paydata;
                 res :=  DBobj.getCard(usrname);
                 return res }
end DB_Wire;

---

Class bookTrip_Wire is
   Signals:
      bookTrip_bell(from:Token,to:Token,out:date,in:date);
```

```
        bookTrip_reply(Reply:bool,fconf:Token, hconf:Token,
                       amount);
        bookTrip_cancel;
        bookTrip_commit;
        bookTrip_revoke;
    State Top =s1
    Transitions:
        s1 -> s1 { bookTrip_bell(from,to,out,in) /
                   BAobj.bookTrip_bell(from,to,out,in) }
        s1 -> s1 { bookTrip_reply(Reply,fconf,hconf,amount) /
                   CRobj.bookTrip_reply(Reply,fconf,
                   hconf,amount) }
        s1 -> s1 { bookTrip_cancel / BAobj.bookTrip_cancel }
        s1 -> s1 { bookTrip_commit / BAobj.bookTrip_commit }
        s1 -> s1 { bookTrip_revoke / BAobj.bookTrip_revoke }
end bookTrip_Wire;
```

COMPONENTS

```
Class BookingAgent is

Signals:
    login_bell(usr: Token,pwd:Token);
    login_cancel;
    login_commit;
    login_revoke;
    bookTrip_bell(from:Token,to:Token,out,in);
    bookTrip_cancel;
    bookTrip_commit;
    bookTrip_revoke;
    bookFlight_reply(Reply:bool,fconf:Token,amount,
                     beneficiary:Token,payService:Token);
    bookHotel_reply(Reply:bool,hconf:Token);
    payment_reply(Reply:bool,proof:Token);
    ackRefundRcv_bell(amount);

Vars:
    s: bookingstatus := START;
    logged: bool := false;
    traveller;
```

```
    travcard;
    bookflight_amount;
    bookflight_beneficiary:Token;
    bookflight_fconf:Token;
    bookflight_amount:Token;
    booktrip_in:date;
    booktrip_out:date;
   KD :=0;

State Top =s1

Transitions:

LoginBell1: s1 -> s1
       { login_bell(usr,pwd) [s = START] /
         logged := DBW.log(usr,pwd);
         if logged then {
            s := LOGGED;
            traveller := DBW.getData(usr);
            travcard := DBW.getCard(usr) }
         else { s := END_UNBOOKED };
        loginW.login_reply(logged);
       }

TripBell1: s1 -> s1
       { bookTrip_bell(from,to,out,in)
           [(s = LOGGED)] /
         booktrip_in := in;
         booktrip_out := out;
         s := QUERIED;
         FlightW.bookFlight_bell(from,to,out,in,traveller);
       }

FlightAnswer1: s1 -> s1
      { bookFlight_reply(Reply,fconf,amount,
                         beneficiary,payService)
      [s=QUERIED] /
      bookflight_amount := amount;
      bookflight_beneficiary := beneficiary;
      bookflight_fconf := fconf;
      bookflight_amount := amount;
      if Reply=True  then {s := FLIGHT_OK}
      else {s := END_UNBOOKED};
```

```
      if Reply=true {
        HotelW.bookHotel_bell(booktrip_out,booktrip_in,
                              traveller) }
      else { bookTripW.bookTrip_reply(False,null,
                                      null,null) };
      }

HotelAnswer1 : s1 -> s1
      { bookHotel_reply(Reply,hconf) [s=FLIGHT_OK] /
      if Reply=True then { s := HOTEL_OK}
      else { s := END_UNBOOKED};
      if Reply=True then {
       bookTripW.bookTrip_reply(Reply,bookflight_fconf,
                                hconf,bookflight_amount) }
      else {bookTripW.bookTrip_reply(False,null,
                                     null,null)};
      }

TripCommit: s1 ->s1
      { bookTrip_commit [s=HOTEL_OK] /
      s := CONFIRMED;
      FlightW.bookFlight_commit;
      HotelW.bookHotel_commit;
      paymentW.payment_bell(bookflight_amount,
                            bookflight_beneficiary,
                            traveller,travcard);
      }

TripCancel1: s1 ->s1
      { bookTrip_cancel [ s= HOTEL_OK] /
        s := END_UNBOOKED;
        FlightW.bookFlight_cancel;
        HotelW.bookHotel_cancel;
      }

PaymentAnswer1: s1 ->s1
      { payment_reply(Reply,proof) [s= CONFIRMED] /
        if Reply=True then { s := END_PAYED}
        else {s :=  END_UNBOOKED};
        payAckW.payAck_bell(proof, Reply);
      }

TripCompensate1: s1 ->s1
```

```
      { bookTrip_revoke [(s= END_PAYED)]   /
        s := COMPENSATING;
        FlightW.bookFlight_revoke;
        HotelW.bookHotel_revoke;
        }

TripRefund: s1 ->s1
      { ackRefundRcv_bell(amount) [s = COMPENSATING] /
       s = END_COMPENSATED;
       ackRefundSndW.ackRefundSnd_bell(amount);
      }

end BookingAgent;

---

Class UserDB is

Operations:
  log(usrname:Token,password:Token):bool;
  getData(usrname: Token): Token;
  getCard(usrname: Token): Token;

Vars:
   cr_usrdata: Token := UserData;
   cr_paydata: Token := PayData;

State Top = s1

Transitions:

Log:  s1 -> s1 {log(usrname,password) / return True}

Log:  s1 -> s1 {log(usrname,password) / return False}

GetData:  s1 -> s1 {getData(usrname) / return cr_usrdata }

GetCard: s1 -> s1  {getCard(usrname) / return cr_paydata }

end UserDB;
```

```
----------REQUIRES-INTERFACES ----------
-----------------------------------------

Class HotelAgent is
Signals:
   lockHotel_bell(checkin:date,checkout:date,traveller);
   lockHotel_commit;
   lockHotel_cancel;
   lockHotel_revoke;

Vars:
   lockHotel_bell_enabled : bool := true;
   lockHotel_commit_enabled : bool := false;
   lockHotel_cancel_enabled : bool := false;
   lockHotel_revoke_enabled : bool := false;

   lockHotel_reply_sent : bool = false;
   lockHotel_bell_executed : bool := true;
   lockHotel_commit_executed : bool := false;
   lockHotel_cancel_executed : bool := false;
   lockHotel_revoke_executed : bool := false;

   lockHotel_checkin:date;
   lockHotel_checkout:date;
   lockHotel_traveller;

   lockHotel_Reply: bool := false;
   hconf: Token := HConf;

State Top = A / B
State A = s1, s2, s3, s4, s5, s6
State B = s1, s2

Transitions:
LockHotel_Bell_1: A.s1 -> A.s2
      { lockHotel_bell(checkin,checkout,traveller)
            [lockHotel_bell_enabled] /
        lockHotel_bell_enabled := false;
        lockHotel_checkin := checkin;
        lockHotel_checkout := checkout;
        lockHotel_Reply := true;
        HotelW.lockHotel_reply(true,hconf);
        lockHotel_reply_sent := true;
```

```
        lockHotel_commit_enabled := true;
        lockHotel_cancel_enabled := true;
       }

LockHotel_Bell_2: A.s1 -> A.s6
       { lockHotel_bell(checkin,checkout,traveller)
            [lockHotel_bell_enabled] /
        lockHotel_bell_enabled := false;
        lockHotel_checkin := checkin;
        lockHotel_checkout := checkout;
        lockHotel_traveller := traveller;
        lockHotel_Reply := false;
        HotelW.lockHotel_reply(false,null);
        lockHotel_reply_sent := true;
       }

LockHotel_Cancel: A.s2 -> A.s3
        { lockHotel_cancel [lockHotel_cancel_enabled] /
        lockHotel_cancel_enabled := false;
        }

LockHotel_Commit: A.s2 -> A.s4
        { lockHotel_commit [lockHotel_commit_enabled] /
        lockHotel_commit_executed := true;
        lockHotel_commit_enabled := false;
        }

lockHotel_Revoke: A.s4 -> A.s5
        { lockHotel_revoke [lockHotel_revoke_enabled] /
        lockHotel_revoke_enabled := false;
        }

rule_1a: B.s1 -> B.s2
       { -
         [lockHotel_commit_executed] /
             lockHotel_revoke_enabled := true;
       }

end HotelAgent;

---

Class FlightAgent is
```

```
Signals:
   lockFlight_bell(from:Token,to:Token,out:date,
                   in:date,traveller:Token);
   lockFlight_cancel;
   lockFlight_commit;
   lockFlight_revoke;
   payAck_bell(proof, status: bool);

Vars:
   Priority :=1;

   lockFlight_bell_enabled: bool := true;
   lockFlight_commit_enabled: bool := false;
   lockFlight_cancel_enabled: bool := false;
   lockFlight_revoke_enabled: bool := false;
   payAck_bell_enabled: bool := false;
   payRefund_bell_active: bool := false;

   lockFlight_reply_sent: bool := false;
   payRefund_bell_sent: bool := false;
   lockFlight_revoke_executed: bool := false;
   payAck_bell_executed: bool := false;
   payRefund_bell_ensured: bool := false;

   lockFlight_from: Token;
   lockFlight_to: Token;
   lockFlight_in: date;
   lockFlight_out: date;
   lockFlight_traveller: Token;
   payAck_proof: bool;
   payAck_status: bool;

   lockFlight_Reply: bool;
   fconf: Token := Fconf;
   amount: int := 10;
   beneficiary: Token := Beneficiary;
   payService: Token := PayService;
   payRefund_amount: int;

State Top = A / B / C / D / E / F

State A = s1, s2, s3, s4, s5, s6
```

```
State B = s1, s2
State C = s1, s2
State D = s1, s2
State E = s1, s2
State F = s1, s2

Transitions:
---- Bell is processed. Reply is true.
LockFlight_Bell_1: A.s1 -> A.s2
      { lockFlight_bell(from,to,out,in,traveller)
            [lockFlight_bell_enabled] /
        lockFlight_bell_enabled := false;
        lockFlight_in := in;
        lockFlight_out := out;
        lockFlight_Reply := true;
        FlightW.lockFlight_reply(lockFlight_Reply,
                                 fconf,amount,beneficiary,
                                 payService);    -- send reply
        lockFlight_reply_sent := true;
        lockFlight_commit_enabled := true;
        lockFlight_cancel_enabled := true;
      }

lockFlight_Bell_2: A.s1 -> A.s6
      { lockFlight_bell(from,to,out,in,traveller)
            [lockFlight_bell_enabled] /
        lockFlight_bell_enabled := false;
        lockFlight_in := in;
        lockFlight_out := out;
        lockFlight_Reply := false;
        FlightW.lockFlight_reply(lockFlight_Reply,
                                 null,null,null,null);
        lockFlight_reply_sent := true;
      }

lockFlight_Cancel: A.s2 -> A.s3
       { lockFlight_cancel [lockFlight_cancel_enabled] /
         lockFlight_cancel_enabled := false;
       }

lockFlight_Commit: A.s2 -> A.s4
       { lockFlight_commit [lockFlight_commit_enabled] /
       lockFlight_commit_enabled := false;
```

```
        }

lockFlight_Revoke: A.s4 ->  A.s5
        { lockFlight_revoke [lockFlight_revoke_enabled] /
        lockFlight_revoke_enabled := false;
        lockFlight_revoke_executed := true;
        }

payAck_Bell_1: B.s1 ->  B.s2
      { payAck_bell(proof,status) [payAck_bell_enabled] /
        payAck_bell_enabled := false;
        payAck_bell_executed := true;
        payAck_proof := proof;
        payAck_status := status;
      }

payRefund_Bell: C.s1 -> C.s2
    { - [payRefund_bell_ensured] /
      refundW.payRefund_bell(payRefund_amount);
      payRefund_bell_sent := true;
    }

rule2: D.s1 -> D.s2
      {- [lockFlight_reply_sent and lockFlight_Reply] /
       payAck_bell_enabled := true;
      }

rule_3a: E.s1 -> E.s2
      { -
        [payAck_bell_executed and payAck_status] /
        lockFlight_revoke_enabled := true;
      }

rule4: F.s1 ->  F.s2
       { -
         [lockFlight_revoke_executed] /
            payRefund_amount := amount;
            payRefund_bell_ensured := true;
       }

end FlightAgent;

---
```

```
Class PayAgent is

Signals:
   payment_bell(amount:Token, beneficiary:Token,
                originator:Token, cardNo:Token);
   payment_cancel;
   payment_commit;
   payment_revoke;

Vars:
   payment_bell_enabled: bool := true;
   payment_commit_enabled: bool := false;
   payment_cancel_enabled: bool := false;
   payment_revoke_enabled: bool := false;

   payment_reply_sent: bool := false;
   payNotify_bell_sent: bool := false;

   payNotify_bell_ensured: bool := false;

   payment_amount: Token;
   payment_beneficiary: Token;
   payment_originator: Token;
   payment_cardNo: Token;

   payment_Reply: bool;
   proof: Token := Proof;
   payNotify_status: bool;

State Top = A / B / C

State A = s1, s2, s3, s4, s5, s6
State B = s1, s2
State C = s1, s2

Transitions:

payment_Bell_1: A.s1 -> A.s2
      { payment_bell(amount,beneficiary,originator,cardNo)
            [payment_bell_enabled] /
        payment_bell_enabled := false;
        payment_amount := amount;
```

```
        payment_beneficiary := beneficiary;
        payment_originator := originator;
        payment_cardNo := cardNo;
        payment_Reply := true;
        paymentW.payment_reply(true,proof);
        payment_reply_sent := true;
        payment_commit_enabled := true;
        payment_cancel_enabled := true;
      }

payment_Bell_2: A.s1 -> A.s6
      { payment_bell(amount,beneficiary,originator,cardNo)
            [payment_bell_enabled] /
        payment_bell_enabled := false;
        payment_amount := amount;
        payment_beneficiary := beneficiary;
        payment_originator := originator;
        payment_cardNo := cardNo;
        payment_Reply := false;
        paymentW.payment_reply(false,null);
        payment_reply_sent := true;
      }

payment_Cancel: A.s2 -> A.s3
       { payment_cancel [payment_cancel_enabled] /
         payment_cancel_enabled := false;
       }

payment_Commit: A.s2 -> A.s4
       { payment_commit [payment_commit_enabled] /
         payment_commit_enabled := false;
       }

payment_Revoke: A.s4 -> A.s5
       { payment_revoke [payment_revoke_enabled] /
         payment_revoke_enabled := false;
       }

payNotify_Bell: B.s1 -> B.s2
    { -
      [payNotify_bell_ensured] /
      payNotifyW.payNotify_bell(payNotify_status);
      payNotify_bell_sent := true;
```

```
    }

rule4: C.s1 -> C.s2
       { -
        [payment_reply_sent] /
         payNotify_status := payment_Reply;
         payNotify_bell_ensured := true;
       }

end PayAgent;
```

CLIENT

```
Class Client is
Signals:
   login_reply(Reply);
   bookTrip_reply(Reply:bool, fconf:Token,
                  hconf:Token, amount);
   payNotify_bell(status:bool);
   refund_bell(amount);

Vars:
   login_bell_active := TT;
   login_commit_active := TT;
   login_cancel_active := TT;
   login_revoke_active := TT;
   bookTrip_bell_active := TT;
   bookTrip_commit_active := TT;
   bookTrip_cancel_active := TT;
   bookTrip_revoke_active := TT;

   username: Token := User;
   password: Token := Passwd;
   from: Token := From;
   to: Token := To;
   out: date := Today;
   in: date := Today;

State Top = s1
```

```
Transitions:

LoginBell: s1 -> s1
        { - [login_bell_active=TT]/
         loginW.login_bell(username,password);
         login_bell_active := FF;
        }

LoginCommit: s1 -> s1
        { - [login_bell_active=TT]/
         loginW.login_commit;
         login_commit_active := FF;
        }

LoginCancel: s1 -> s1
        { - [login_cancel_active=TT]/
         loginW.login_cancel;
         login_cancel_active := FF;
        }

LoginRevoke: s1 -> s1
        { - [login_revoke_active=TT]/
         loginW.login_revoke;
         login_revoke_active := FF;
        }

BookTripBell: s1 -> s1
        { - [bookTrip_bell_active=TT]/
         bookTripW.bookTrip_bell(from,to,out,in);
         bookTrip_bell_active := FF;
        }

BookTripCommit: s1 -> s1
        { - [bookTrip_commit_active=TT]/
         bookTripW.bookTrip_commit;
         bookTrip_commit_active := FF;
        }

BookTripCancel: s1 -> s1
        { - [bookTrip_cancel_active=TT]/
         bookTripW.bookTrip_cancel;
         bookTrip_cancel_active := FF;
        }
```

```
BookTripRevoke: s1 -> s1
        { - [bookTrip_revoke_active=TT]/
         bookTripW.bookTrip_revoke;
         bookTrip_revoke_active := FF;
        }

end Client;
```

DATA TYPES

```
Class bookingstatus is end;
Class usrdata is end;
Class paydata is end;
Class date is end;


Objects:
```

SERVICE MODULE

```
BAobj: BookingAgent;
HAobj: HotelAgent;
FAobj: FlightAgent;
CRobj: Client;
PAobj: PayAgent;
DBobj: UserDB;
payAckW: payAck_Wire;                -- BA -> FA
refundW: refund_Wire;                -- FA -> BA
payNotifyW: payNotify_Wire;          -- PA -> CR
ackRefundSndW: ackRefundSnd_Wire     -- FA -> CR
HotelW: Hotel_Wire;                  -- BA <-> HA
FlightW: Flight_Wire;                -- BA <-> FA
paymentW: payment_Wire;              -- BA <-> PA
loginW: login_Wire;                  -- CR <-> BA
DBW: DB_Wire;                        -- BA <-> DB
bookTripW: bookTrip_Wire;            -- CR <-> BA
```

```
-----------------------------------------------------------
-----------------DATA OBJECTS------------------------------
-----------------------------------------------------------

START, LOGGED, QUERIED, FLIGHT_OK, HOTEL_OK, CONFIRMED,
  END_PAYED,END_UNBOOKED, COMPENSATING,
  END_COMPENSATED: bookingstatus;

TT,FF,XX, ACTIVE: Token;

Today, Tomorrow, AfterTomorrow: date;
HConf, User, Passwd, UserData, PayData, From, To: Token;
Fconf, Beneficiary,PayService, Proof: Token;
```

Bibliography

[1] J. Abreu, L. Bocchi, J. L. Fiadeiro, and A. Lopes. Specifying and composing interaction protocols for service-oriented system modelling. In *Formal Techniques for Networked and Distributed Systems*, volume 4574 of *LNCS*, pages 358–373. Springer, 2007.

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

[3] J. Abreu, F. Mazzanti, J. L. Fiadeiro, and S. Gnesi. A model-checking approach for Service Component Architectures. In *Formal Techniques for Distributed Systems*, volume 5522 of *LNCS*, pages 219–224. Springer, 2009.

[4] 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.

[5] F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. *PICCOLA—a small composition language*. Cambridge University Press, New York, USA, 2001.

[6] E. Al-Masri and Q. H. Mahmoud. Investigating web services on the world wide web. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 795–804, New York, NY, USA, 2008. ACM.

[7] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of msc graphs. *Theoretical Computer Science*, 331(1):97–114, 2005.

[8] 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.

[9] M. 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 *FMICS'07*, LNCS. Springer-Verlag, Berlin, 2007.

[10] M. Beek, S. Gnesi, F. Mazzanti, and C. Moiso. Formal Modelling and Verification of an Asynchronous Extension of SOAP. In *ECOWS'06*, pages 287–296. IEEE Computer Society, Los Alamitos, CA, 2006.

[11] M. Beisiegel et al. Service Component Architecture — White Paper, 2005.

[12] M. Beisiegel et al. Service Component Architecture Specifications, 2007.

[13] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American Magazine*, May 2001.

[14] 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. Submitted., 2009.

[15] L. Bocchi, J. L. Fiadeiro, and A. Lopes. Service-oriented modelling of automotive systems. In *The 32nd Annual IEEE International on Computer Software and Applications, COMPSAC'08*, pages 1059–1064. IEEE, 2008.

[16] L. Bocchi, J. L. Fiadeiro, and A. Lopes. A use-case driven approach to formal service-oriented modelling. In *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science*, pages 155–169. Springer, 2008.

[17] L. Bocchi, Y. Hong, A. Lopes, and J. L. Fiadeiro. From BPEL to SRML: a formal transformational approach. In *Web Services and Formal Methods*, volume 4937 of *LNCS*, pages 92–107. Springer, 2008.

[18] D. Booth et al. Web services architecture, 2004. http://www.w3.org/TR/ws-arch/.

[19] M. Boreale, R. Bruni, L. Caires, R. De 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.

[20] R. Bruni, I. Lanese, H. Melgratti, and E. Tuosto. Multiparty sessions in SOC. In *Coordination Models and Languages*, 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 *Algebraic Methodology and Software Technology*, 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*, 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] M. Carbone, K. Honda, and N. Yoshida. Theoretical aspects of communication-centred programming. *Electronic Notes in Theoretical Computer Science*, 209:125 – 133, 2008.

[26] R. Chinnici et al. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, 2007. http://www.w3.org/TR/wsdl20/.

[27] L. Clement et al. UDDI Version 3 specification, 2002. http://www.uddi.org/pubs/uddi-v3.00-published-20020719.htm.

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

[29] G. Denaro and M. Pezzè. Petri nets and software engineering. In *Lectures on Concurrency and Petri Nets*, pages 439–466. Springer, 2003.

[30] 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.

[31] J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Verification of computation orchestration via timed automata. In *Formal Methods and Software Engineering*, volume 4260 of *LNCS*, pages 226–245. Springer, 2006.

[32] 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*, pages 411–420. ACM, 1999.

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

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

[35] D. Fensel, M. Kifer, and J. de Bruijn. WSML v1.0 Specification, 2008. http://www.wsmo.org/wsml/wsml-syntax.

[36] J. L. Fiadeiro, A. Lopes, and L. Bocchi. A formal approach to Service Component Architecture. *Web Services and Formal Methods*, 4184:193–213, 2006.

[37] J. L. Fiadeiro, A. Lopes, and L. Bocchi. Algebraic semantics of service component modules. In *Recent Trends in Algebraic Development Techniques*, volume 4409 of *LNCS*, pages 37–55. Springer, 2007.

[38] J. L. Fiadeiro, A. Lopes, and L. Bocchi. An abstract model of service discovery and binding. Submitted. Available from www.cs.le.ac.uk/people/jfiadeiro, 2008.

[39] J. L. Fiadeiro, A. Lopes, L. Bocchi, and J. Abreu. A formal approach to service-oriented modelling. Presented at the 9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services, 2009.

[40] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proceedings of the 13th international conference on World Wide Web*, pages 621–630. ACM, 2004.

[41] 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.

[42] C. Harding et al. SOA Source Book, 2006. http://www.opengroup.org/projects/soa/.

[43] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In *Automata, Languages and Programming*, volume 85, pages 299–309. Springer, 1980.

[44] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri nets. In *Business Process Management*, volume 3649 of *LNCS*, pages 220–235. Springer, 2005.

[45] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

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

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

[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] D. Kitchin, A. Quark, W. R. Cook, and J. Misra. The Orc programming language. In *Formal Techniques for Distributed Systems*, volume 5522 of *LNCS*, pages 1–25. Springer, 2009.

[50] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Programming Languages and Systems*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.

[51] R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL. In *Journal of Logic and Algebraic Programming*. Elsevier press, 2005.

[52] M. MacKenzie et al. Reference model for service oriented architecture 1.0, 2006. http://www.oasis-open.org/committees/soa-rm/.

[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 *Fundamental Approaches to Software Engineering*, volume 3442 of *LNCS*, pages 19–33. Springer, 2005.

[55] D. Martin et al. OWL-S: Semantic Markup for Web Services 1.2, 2006. http://www.daml.org/services/owl-s/.

[56] F. Mazzanti. UMC User Guide v3.3. Technical Report 2006-TR-33, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR. Available from http://fmt.isti.cnr.it/WEBPAPER/UMC-UG33.pdf, 2006.

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

[58] 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.

[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*, ACM International Conference Proceeding Series, pages 77–88. ACM, 2002.

[60] Object Management Group. UML Version 2.2 Specification, 2009. http://www.omg.org/spec/UML/2.2/.

[61] 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.

[62] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 1997.

[63] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*. Springer, 1998.

[64] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3), 2001.

[65] Telecommunication Standardization Sector of ITU. Specification and Description Language (Z.100 series ITU Recs.), 1999.

[66] W. van der Aalst and M. Pesic. DecSerFlow: Towards a truly declarative service flow language. In *Web Services and Formal Methods*, volume 4184, pages 1–23. Springer, 2006.

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

[68] V. T. Vasconcelos. TyCO gently. DI/FCUL TR 01–4, DIFCUL, July 2001.

[69] H. T. Vieira, L. Caires, , and J. C. Seco. The Conversation Calculus: A model of service-oriented computation. In *Programming Languages and Systems*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.

[70] M. Weidlich, G. Decker, and M. Weske. Efficient analysis of BPEL 2.0 processes using pi-calculus. In *The 2nd IEEE Asia-Pacific Service Computing Conference*, pages 266–264. IEEE, 2007.

[71] M. Wirsing et al. SENSORIA: A systematic approach to developing service-oriented systems — white paper, 2007.

[72] X. Yi and K. J. Kochut. A CP-nets-based design and verification framework for web services composition. *IEEE International Conference on Web Services*, 2004.