

The Sensoria Reference Modelling Language

José Fiadeiro¹, Antónia Lopes², Laura Bocchi¹, and João Abreu¹

¹ Department of Computer Science, University of Leicester

² Department of Informatics, Faculty of Sciences, University of Lisbon

Abstract. This chapter provides an overview of SRML — the SENSORIA Reference Modelling Language. SRML offers a technology-agnostic framework in which business services and activities can be modelled independently of the languages in which components are implemented and the network protocols through which they communicate. From a methodological point of view, SRML supports Service-Oriented Computing as a new paradigm in which computations result from a distributed orchestration of software components and external services that are procured on the fly subject to a negotiation of service level agreements (SLAs). Our focus will be on the language primitives that SRML offers for orchestrating business services and activities, defining the interfaces through which services are offered or procured, and expressing the SLA constraints that apply to service provision. We also present elements of the mathematical semantics that underpins the modelling approach, and the way it supports qualitative and quantitative analysis.

1 Introduction

This chapter provides an overview of the modelling language — SRML — that we developed in SENSORIA. We present the language primitives that SRML offers for modelling business services and activities, and discuss the methodological approach that SRML supports, which includes the use of the UMC model-checker (developed at CNR-ISTI) for qualitative analysis and of the Markovian process algebra PEPA (developed at the University of Edinburgh) for quantitative analysis of timing properties. Only some elements of the mathematical semantics that we developed for the approach are provided in this chapter; full details can be found in [4,6,32,33,34].

Our approach addresses *Service-Oriented Computing* (SOC) as a new computational paradigm in which interactions are no longer based on fixed or programmed exchanges between specific parties — what is known as clientship in object-oriented programming — but on the provisioning of *services* by external providers that are procured on the fly subject to a negotiation of service level agreements (SLAs). In SOC, the processes of discovery and selection of services are not coded (at design time) as part of the applications that implement business activities, but performed by the middleware according to functional and non-functional requirements (SLAs). We set ourselves to address the challenge

raised on software engineering methodology by the need of declaring such requirements as part of the models of service-oriented applications, reflecting the business context in which services and activities are designed.

A number of research initiatives have been proposing formal approaches that address different aspects of SOC independently of the specific languages that organisations such as OASIS (www.oasis-open.org) and W3C (www.w3.org) are making available for Web Services. For example, as presented in other chapters of this book, several service calculi (e.g. [30,38]) have been developed in SENSORIA that address operational foundations of SOC (in the sense of how services compute) by providing a mathematical semantics for the mechanisms that support ‘coreography’ or ‘orchestration’ — sessions, message/event correlation, compensation, inter alia. Whereas such calculi address the need for specialised language primitives for *programming* in this new paradigm, they are not abstract enough to understand the *engineering* foundations of SOC, i.e. those aspects (both technical and methodological) that concern the way applications can be developed to provide business solutions, independently of the languages in which services are programmed.

This is why, in defining SRML, we used as a source of inspiration the Service Component Architecture (SCA) [2]. SCA makes available a general assembly model and binding mechanisms for service components and clients that may have been programmed in possibly many different languages, e.g. Java, C++, BPEL, or PHP. However, where SCA supports bottom-up low-level design, our aim for SRML was, instead, to address top-down high-level design. More specifically, our aim was to develop models and mechanisms that support the design of complex services from business requirements, and analysis techniques through which designers can verify or validate properties of composite services that can then be put together from (heterogeneous) service components using assembly and binding techniques such as the ones provided by SCA. This shift of emphasis from programming to (business) modelling, from component interoperability to business integration, implies that we will be discussing SOC at a level of abstraction that is different from most other work on Web services (e.g. [11,44] or Grid computing (e.g. [35])).

Having this in mind, the chapter proceeds as follows. In Section 2, we provide an overview of the ‘engineering’ architecture and processes that we see supporting SOC in Global Computing. In Section 3, we provide a brief overview of how we support the transition from business requirements to high-level design models using a (service-oriented) extension of use-case diagrams. In Section 4, we put forward the coordination model on which SRML is based. In Section 5, we present the modelling primitives of SRML. In Section 6, we discuss our model of configuration management. In Section 7, we discuss the use of model-checking techniques for analysing functional properties of complex services. Finally, in Section 8, we discuss the use of the Markovian process algebra PEPA for analysing timing properties. In Appendix A, we collect the different icons used in the graphical representation of modules. As a running example, we will use a mortgage-brokerage service that is part of the financial case study developed

by SENSORIA. The full specification of the service module GETMORTGAGE is presented in Appendix B. Although our approach is formal, in the sense that a mathematical semantics is available for all the primitives of the language [4,33], the paper is mostly mathematics-free with the exception of Sections 4.3, 6, 7.1 and 8.

2 Engineering Software for Service-Overlay Computers

The term ‘service’ is being used in a wide variety of contexts, often with different meanings. In SENSORIA, we address the notion of ‘service-overlay computer’, by which we mean the development of highly-distributed loosely-coupled applications over ‘global computers’ (GC) — “computational infrastructures available globally and able to provide uniform services with variable guarantees for communication, cooperation and mobility, resource usage, security policies and mechanisms” [1].

In this setting, there is a need to rethink the way we engineer software applications, moving from the typical ‘static’ scenario in which components are assembled to build a (more or less complex) system that is delivered to a customer, to a more ‘dynamic’ scenario in which (smaller) applications are developed to run on such global computers and respond to business needs by interacting with services and resources that are globally available. In this latter setting, there is much more scope for flexibility in the way business is supported: business processes can be viewed globally as emerging from a varying collection of loosely-coupled applications that can take advantage of the availability of services procured on the fly when they are needed.

In this context, the notion of ‘system’ itself, as it applies to software, also needs to be revisited. If we take one of the accepted meanings of system — *a combination of related elements organised into a complex whole* — we can see why it is not directly applicable to SOC/GC: services get combined at run time and redefine the way they are organised as they execute; no ‘whole’ is given *a priori* and services do not compute within a fixed configuration of a ‘universe’. In a sense, we are seeing reflected in software engineering the trend for ‘globalisation’ that is now driving the economy.

SOC brings to the front many aspects that have already been discussed about component-based development (CBD), for instance in [25]. Given that different people have different perceptions of what SOC and CBD are, we will simply say that, in this paper, we will take CBD to be associated with what we called the static engineering approach. For instance, starting from a universe of (software) components as structural entities, Broy et al view a service as a way of orchestrating interactions among a subset of components in order to obtain some required functionality — “services coordinate the interplay of components to accomplish specific tasks” [17]. As an example, we can imagine that a bank will have available a collection of software components that implement core functionalities such as computing interests or charging commissions, which can be used in different products such as savings or loans.

SOC differs from this view in that there is no such fixed system of components that services are programmed to draw from but, rather, an evolving universe of software applications that service providers publish so that they can be discovered by (and bound to) business activities as they execute. For instance, if documents need to be exchanged as part of a loan application, the bank may rely on an external courier service instead of imposing a fixed one. In this case, a courier service would be discovered for each loan application that is processed, possibly taking into account the address to which the documents need to be sent, speed of delivery, reliability, and so on. However, the added flexibility provided through SOC comes at a price — dynamic interactions impose the overhead of selecting the co-party at each invocation — which means that the choice between invoking a service and calling a component is a decision that needs to be taken according to given business goals. This is why SRML makes provision for both SOC and CBD types of interaction (through *requires* and *uses* interfaces as discussed in Section 3).

To summarise, the impact that we see (and explore) SOC to have on software engineering methodology stems from the fact that applications are built without knowing who will provide services that may be required, and that the discovery and selection of such services is performed, on the fly, by dedicated middleware components. This means that application developers cannot rely on the fact that someone will interact with them to implement the services that may be required so as to satisfy their requirements. Therefore, service-oriented ‘clientship’ needs to be based on shared ontologies of data and service provision. Likewise, service development is not the same as developing software applications to a customer’s set of requirements: it is a separate business that, again, has to rely on shared ontologies of data and service provision so that providers can see their services discovered and selected.

This view is summarised in Figure 1, where we elaborate beyond the basic Service-Oriented Architecture [8] to make explicit the different stakeholders and the way they interact, which is important for understanding the formal model that we are proposing. In this model, we distinguish between ‘business activities’ and ‘services’ as software applications that pertain to different stakeholders (see [36] for a wider discussion on the stakeholders of service-oriented systems):

- *Activities* correspond to applications developed according to requirements provided by a business organisation, e.g. the applications that, in a bank, implement the financial products that are made available to the public. The *activity repository* provides a means for a run-time engine to trigger such applications when the corresponding requests are published, say when a client of the bank requests a loan at a counter or through on-line banking. Activities may be implemented over given components (for instance, a component for computing and charging interests) in a traditional CBD way, but they can also rely on services that will be procured on the fly using SOC (for instance, an insurance for protecting the customer in case he/she is temporarily prevented from re-paying the loan due to illness or job loss). In SRML, activities are modelled through *activity modules*. As discussed in

Section 3, these identify the components that activities need to be bound to when they are launched and the services (types) that they may require as they execute. Activity modules also include a specification of the workflow that orchestrates the interactions among all the parties involved in the activity and a number of SLA constraints used for negotiating service provision from external parties.

- *Services* differ from activities in that they are not developed to satisfy specific business requirements of a given organisation but to be published (in service repositories) in ways that allow them to be discovered when a request for an external service is published in the run-time environment. As such, they are classified according to generic service descriptions — what in Section 5.1 we call ‘business protocols’ — that are organised in a hierarchical ontology to facilitate discovery. Services are typed by *service modules*, which, like activity modules, identify the components and additional services that may be required together with a specification of the workflow that orchestrates the interactions among them so as to deliver the properties declared in the service descriptions — their ‘provides-interfaces’. Service modules also specify service-level agreements that need to be negotiated during matchmaking and selection.

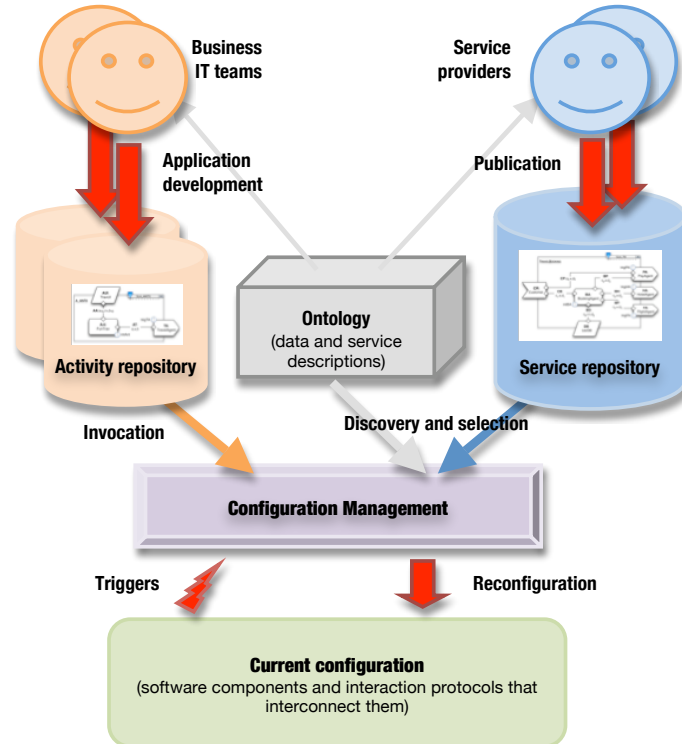


Fig. 1. Overall ‘engineering’ architecture and processes.

- The *configuration management* unit (discussed in Section 6) is responsible for the binding of the new components and connectors that derive from the instantiation of new activities or services. A formal model can be found in [33].
- The *ontology* unit is responsible for organising both data and service descriptions. In this paper, we do not discuss the classification and retrieval mechanisms per se. See, for instance, [39,45] for some of the aspects involved when addressing such issues.

Notice that the ‘business IT teams’ and the ‘service providers’ can be totally independent and unrelated: the former are interested in supporting the business of their companies or organisations, whereas the latter run a business of their own. They can also belong to the same organisation, as illustrated in our case study. In both cases, they share the ontology component of the architecture so that they can do business together.

3 From Use-Case Diagrams to SRML Modules

Before we introduce the modelling primitives that SRML offers for high-level (business) design, it is important to show how traditional use-case diagrams can be extended so as to support the engineering approach that we described in Section 2. In order to illustrate our approach, we consider the (simplified) case of a financial services organisation that wants to offer a mortgage-brokerage service GETMORTGAGE. This service involves the following steps:

- Proposing the best mortgage deal to the customer that invoked the service;
- Taking out the loan if the customer accepts the proposal;
- Opening a bank account associated with the loan if the lender does not provide one;
- Getting insurance if required by either the customer or the lender.

In our example, the selection of a lender is restricted to firms that are considered to be reliable. For this reason, we consider an UPDATEREGISTRY activity supporting the management of a registry of reliable lenders. This activity relies on an external certification authority that may vary according to the identity of the lender.

3.1 Use-case diagrams for service-oriented modelling

Traditionally, use-case diagrams are used for providing an overview of usage requirements for a system that needs to be built. As discussed in Section 2, and reporting to Figure 1, our aim is to address a novel development process that does not aim at the construction of a ‘system’ but, rather, of two kinds of software applications — services and activities. Activities are bound to other software components statically (in a component-based way) and services are bound dynamically (in a service-oriented way). Services and activities have the

particularity that each has a single usage requirement. Hence, they can be perceived as use cases. On the other hand, from a business point of view, two or more services and activities developed can belong to one logical unit.

In our example, UPDATEREGISTRY should be treated as an activity in the sense that it is driven by the requirements of the financial services organisation itself — it will be stored in an activity repository and will be invoked by internal applications (e.g., a terminal interface). On the other hand, GETMORTGAGE is meant to be placed in a service repository for being discovered and bound to activities or services running ‘globally’, i.e. not necessarily in the financial services organisation.

Both UPDATEREGISTRY and GETMORTGAGE can be seen to operate as part of a same business unit and, hence, it makes sense to group them in the same use-case diagram — use-case diagrams are useful for structuring usage requirements of units of business logic. In order to reflect the methodological implications of our approach, we propose a number of extensions to the standard notation of use cases. Figure 2 uses the mortgage example to illustrate our proposal: the diagram represents a business logical unit with the two use cases identified before. The rectangle around the use cases, which in traditional use-case diagrams indicates the boundary of the system at hand, is used to indicate the scope of the business unit. Anything within the box represents functionality that is in scope and anything outside the box is considered not to be in scope.

For the UPDATEREGISTRY activity, the primary actor is *Registry Manager*; its goal is to control the way a registry of trusted lenders is updated. The registry itself is regarded as a supporting actor. The *Certification Authority* on which UPDATEREGISTRY relies is also considered a supporting actor in the use case because it is an external service that needs to be discovered based on the nature of the lender being considered.

In the GETMORTGAGE service, the primary actor is a *Customer* that wants to obtain a mortgage. The use case has four supporting actors: *Lender*, *Bank*, *Insurance* and *Registry*. The *Lender* represents the organisation (e.g., a bank or building society) that lends the money to the customer. Because only reliable firms can be considered for the selection of the lender, the use case involves communication with *Registry*. When the lender does not provide a bank account, the use case involves an external *Bank* for opening a new account. Similarly, the use case involves interaction with an *Insurance* provider for situations where the lender requires insurance or the customer decides to get one.

As in traditional use cases, we view an actor as any entity that is external to the business unit and interacts with at least one of its elements in order to perform a task. As motivated above, we can distinguish between different kinds of actors, which led us to customise the traditional icons as depicted in Figure 2. These allow us to discriminate between *user/requester* and *resource/service* actors. *User-actors* and *requester-actors* are similar to primary actors in traditional use-case diagrams in the sense that they represent entities that initiate the use case and whose goals are fulfilled through the successful completion of the use case. The difference between them is that a *user-actor* is a role played by an

entity that interacts with the activity, while a *requester-actor* is a role played by one or more software components operating as part of the activity that triggers the discovery of the service.

For instance, the user-actor *Registry Manager* represents an interface for an employee of the business organisation that is running *Mortgage Finder* whereas the requester-actor *Customer* represents an interface for a service requester that can come from any external organisation. A requester-actor can be regarded as an interface to an abstract user of the functionality that is exposed as a service; it represents the range of potential customers of the service and the requirements typically derive from standard service descriptions stored in service repositories such as the UDDI. In SRML, and reporting to Figure 1, these descriptions are given by business protocols (discussed in Section 5.1) and organised in a shared ontology, which facilitates and makes the discovery of business partners more

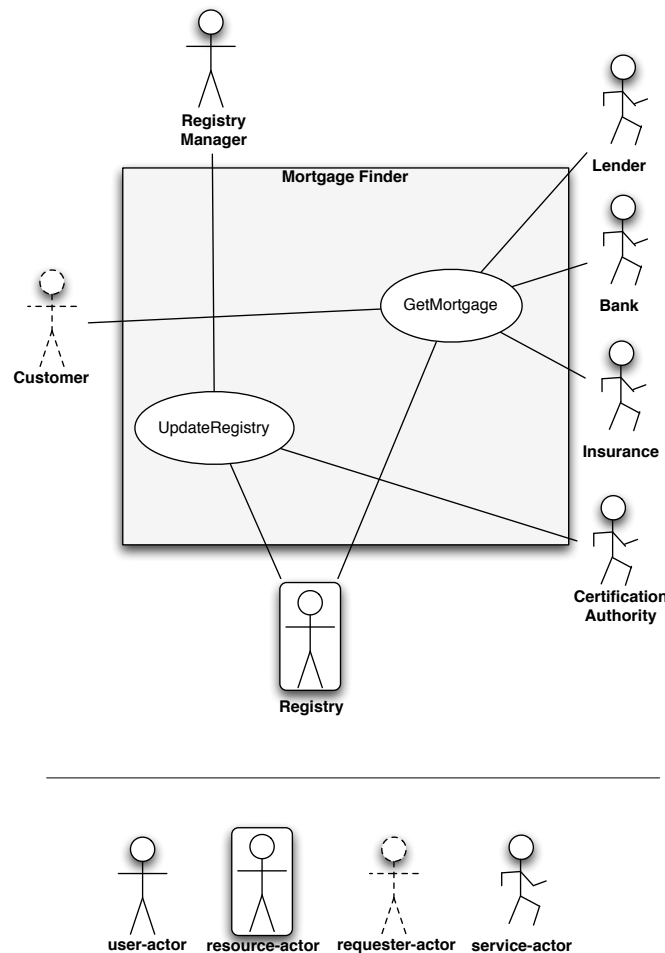


Fig. 2. Service-oriented use-case diagram for Mortgage Finder.

effective. The identification of requester-actors may take advantage of existing descriptions in the ontology or it may identify new business opportunities. In this case, the ontology would be extended with new business protocols corresponding to the new types of service.

Resource-actors and service-actors of a use case are similar to supporting actors in traditional use-case diagrams in the sense that they represent entities to rely on in order to achieve the underlying business goal. The difference is that a service-actor represents an outsourced functionality to be procured on the fly and, hence, will typically vary from one instance of the use case to another, whereas a resource-actor is an entity that is statically bound and, hence, is the same for all instances of the use case. Resource-actors are typically persistent sources/repositories of information. In general, they are components that are already available to be shared within a business organisation.

The user- and resource-actors, which we represent at the top and bottom of our specialised use-case diagrams, respectively, correspond in fact to the actors that are typically presented on the left and right-hand side in traditional use-case diagrams, respectively. In contrast, the horizontal dimension of the new diagrams, comprising requester- and service-actors, captures the types of interactions that are specific to SOC.

We assume that every use case corresponds to a service-oriented artefact and that the association between a primary actor and a use case represents an instantiation/invoke. For this reason, in this context, we constrain every use case to be associated with only one primary actor (either a requester or a user).

3.2 Deriving the structure of SRML modules

The proposed specialisations of use-case diagrams allow us to identify and derive a number of aspects of the structure of SRML modules — the main modelling primitives that we use for services and activities. Each use case, representing either a service or an activity, gives rise to a SRML service module or activity module, respectively. Figure 3 presents the structure of the modules derived from the use-case diagram in Figure 2.

A SRML module provides a formal model of a service or activity in terms of a configuration of ‘interfaces’ (formal specifications) to the parties involved. In the case of activity modules:

- A *serves-interface* (at the top-end of the module) identifies the interactions that should be maintained between the activity and the rest of the system in which it will operate. This interface results from the user-actor of the corresponding use case.
- *Uses-interfaces* (at the bottom-end of the module) are defined for those (persistent) components of the underlying configuration that the activity will need to interact with once instantiated. These interfaces result from the resource-actors of the corresponding use case and provide formal descriptions of the behaviour required of the actual interfaces that need to be set up for the activity to interact with components that correspond to (persistent) business entities.

- *Requires-interfaces* (on the right-hand boundary of the module) are defined for services that the activity will have to procure from external providers if and when needed. Typically, these reflect the structure of the business domain itself in the sense that they reflect the existence of business services provided outside the scope of the local context in which the activity will operate. These interfaces result from the service-actors of the corresponding use case.
- *Component* and *wire-interfaces* (inside the module) are defined for orchestrating all these entities (actors) in ways that will deliver stated user requirements through the serves-interface. These interfaces are not derived from the use-case diagram but from the description of the corresponding business requirements, i.e. they result from a design step. Typically, a designer will choose pre-defined patterns of orchestration that reflect business components that will be created in support of the activity or chosen from a

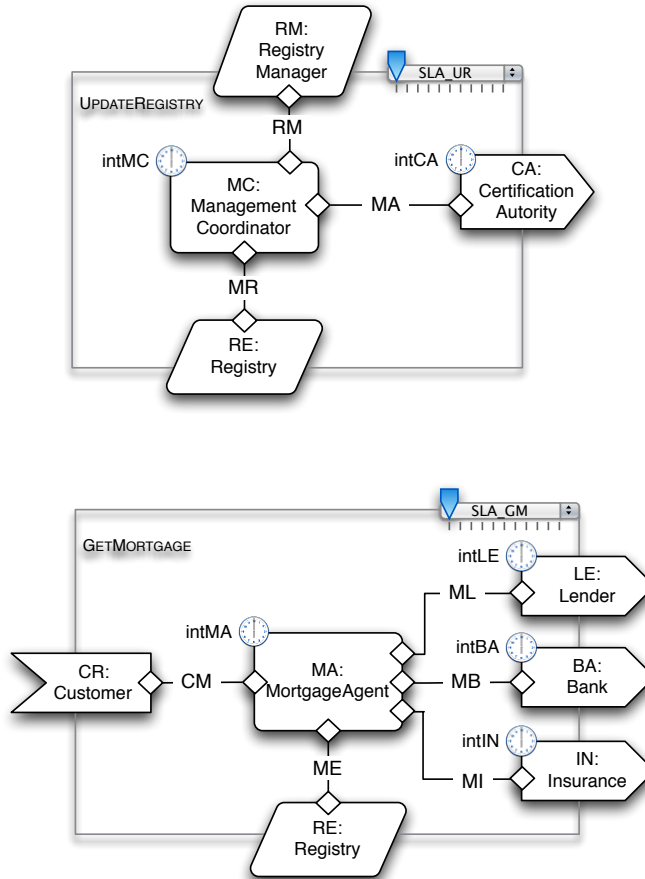




Fig. 3. The SRML modules for the activity **UPDATERegistry** and the service **GETMORTGAGE**.

portfolio of components already available for reuse within the business organisation. The choice of the internal architecture of the module (components and wires) should also reflect the nature of the business communication and distribution network over which the activity will run.

In the case of a service module, a similar diagrammatic notation is used except that a *provides-interface* is used instead of a *serves-interface*:

- The provides-interface should be chosen from the hierarchy of standard business protocols (i.e., descriptions of the same type of those used for requires-interfaces) because the purpose here is to make the service available to the wider market, not to a specific client. It derives from the requester-actor of the corresponding use case.
- Some of the component interfaces will correspond to standard components that are part of the provider’s portfolio. For instance, these may be application-domain dependent components that correspond to typical entities of the business domain in which the service provider specialises.
- Uses-interfaces should be used for those components that the service provider has for insuring persistence of certain effects of the services that it offers.

In addition, both activity and service modules include:

- An internal configuration policy (indicated by the symbol ) , which identifies the triggers of the external service discovery process as well as the initialisation and termination conditions of the components that instantiate the component-interfaces.
- An external configuration policy (indicated by the symbol ) , which consists of the variables and constraints that determine the quality profile of the activity to which the discovered services need to adhere.

The language primitives that are used in SRML for defining all these interfaces as well as the configuration policies are detailed in Section 5. A summary of the graphical notation can be found in Appendix A at the end of the paper.

4 The Coordination Model

The interfaces of a SRML module identified through a use-case diagram reflect business dependencies of services or activities, not the interfaces that software components offer to be interconnected: modules are not models of components but of business processes. In this section, we detail the coordination model that SRML adopts for component interconnection, i.e. we address the nature of the interfaces that components offer and the way wires interconnect them. We also outline a formalisation of this model, full details of which are available from [4].

4.1 Conversational interactions

Typically, in CBD, one organises component interfaces (what they offer to and expect from the rest of the system) in ports, which include the protocols that

regulate message exchange at those ports. In SRML, we have fixed the nature of the interactions and protocols followed by components and wires. We distinguish the following types of interactions:

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

Interactions involve two parties and are described from the point of view of the party in which they are declared, i.e. ‘receive’ means invocations received by the party and sent by the co-party, and ‘send’ means invocations made by the party. Interactions can be synchronous, implying that the party waits for the co-party to reply or complete, or asynchronous, in which case the party does not block. Typically, synchronous (blocking) interactions (i.e., **ask**, **rpl**, **tll** and **prf**) occur with persistent components, reflecting interconnections based on the exchange of *products* (clientship as in OO). The interactions among the components responsible for the orchestration and those involving external services are typically asynchronous (non-blocking, i.e., **r&s**, **s&r**, **snd** and **rcv**) so that the parties can engage in multiple, concurrent conversations. Interactions of type **r&s** and **s&r** are conversational (what we call 2-way), i.e. they involve a number of events exchanged between the two parties:

interaction 🔔	The event of initiating <i>interaction</i> .
interaction 📧	The reply-event of <i>interaction</i> .
interaction ✓	The commit-event of <i>interaction</i> .
interaction ✕	The cancel-event of <i>interaction</i> .
interaction 🔔	The revoke-event of <i>interaction</i> .

The initiation-event is the only event that can be associated also to 1-way asynchronous interaction types (i.e., **snd** **rcv**). The reply-event is sent by the co-party, offering a deal or declining to offer one; in the first case, the party that initiated the conversation may either commit to the deal or cancel the interaction; after committing, the party can still revoke the deal, triggering a compensation mechanism. Every 2-way interaction has an associated *pledge* — a condition that is guaranteed to hold from the moment a positive reply-event occurs until either the commit-event, the cancel-event or the expiration time occurs, whichever happens first. We denote this condition by *interaction*🔔. See Figure 4 for some of the possible scenarios (explained further below).

All interactions can have parameters for transmitting data when they are initiated — declared as \triangleleft . Conversational interactions can also have parameters for carrying a reply — declared as \boxtimes — or for carrying data if there is a commit, a cancel or a revoke — declared as \checkmark , \times and \ddagger , respectively. In particular, every reply-event *interaction* \boxtimes has two distinguished parameters:

- *Reply* is a Boolean parameter that indicates whether the reply is positive, meaning that the co-party is ready to proceed. The value of *interaction.Reply* is *False* if, for some reason related with the business logic, the request *interaction* \triangleleft cannot be fulfilled.
- *UseBy* is a parameter that, in the case of a positive reply, indicates the deadline for receiving the commit and cancel events. The value of this parameter is an expiration time (including the value $+\infty$) obtained by adding the value of the configuration variable (non-functional attribute) *interaction* \bullet^* to the instant at which *interaction* \boxtimes is sent. As discussed in Section 5.2, configuration variables can be subject to negotiation during the discovery/selection process.

Interactions can be seen as ports in the traditional CBD sense, the associated events representing the interface of the components. The sequence diagrams in Figure 4 illustrate the protocol that is associated with every interaction for which the reply is positive. 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 *interaction* \checkmark (this can however be constrained by the business logic, for instance, by defining a deadline for compensation). In the middle, there is a cancellation; in this situation, a revoke is not available. In the case on the right, the expiration time occurs without a commit or cancel having occurred; this implies that no further events for that interaction will occur. In Section 5, we give examples of the intended usage of these primitives.

Events occur during state transitions in both parties involved in the interaction: we use *event!* in order to refer to the publication of event in the life of the initiating party, and *event?* (resp. *event!*_i) for its execution (resp. being discarded) by the party that receives it. The occurrences of *event!* and *event?* (or

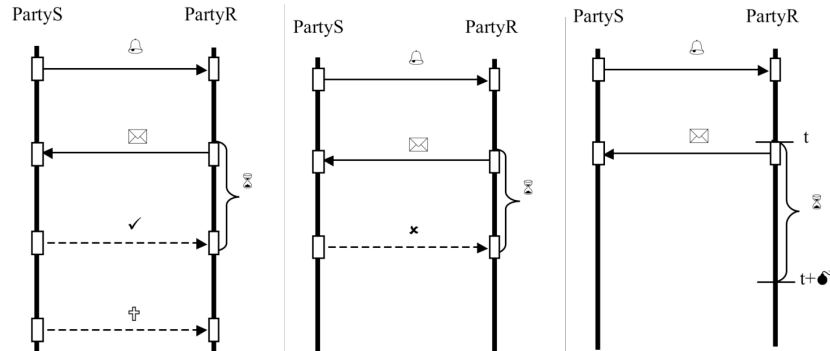


Fig. 4. The protocol of 2-way interactions when the reply is positive.

$event_i$) may not coincide in time: we consider that there may exist a delay between publishing and delivering an event. The value of this delay is given by the configuration variable *Delay* associated with the wire through which the events are transmitted (see Figure 5). In Section 8, we explore timing aspects of service provision in more detail, including the use of PEPA [37] for stochastic analysis.

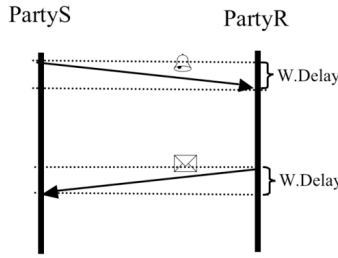


Fig. 5. The intuitive semantics of delays.

4.2 Deriving interactions from message sequence diagrams

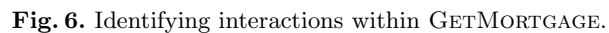
One of the ways that we have found useful for identifying the interactions that are relevant for defining a given activity or service module is to draw message sequence diagrams that characterise the interconnections required between the different parties. For instance, the message sequence diagram in Figure 6 depicts the workflow that is initiated by the initial request received by GETMORTGAGE from the customer *CR*.

4.3 A formal model

The overall coordination model of SRML can be summarised as follows (see [4] for details). We work over configurations of global computers defined by a set **COMP** of components (applications deployed over execution platforms) linked through wires (e.g. interconnections between components over a given communication network), the set of which we denote by **WIRE**.

A *state* consists of:

- The set *PND* of the events that are pending in the wires, i.e. the events that have been published but not yet delivered by the wires to the corresponding co-parties;
- The set *INV* of the events that have been invoked, i.e. those that were delivered by the wires and are stored locally by the components that received them, waiting to be processed;
- The time at that state;
- The set of pledges that hold in that state;
- A record of all events that have been published (!), delivered (i), executed (?) or discarded (i);



- In this model, state transitions are characterised by what we call a *computation step*, consisting of:

- An ordered pair of states SRC (source) and TRG (target);
- A subset DLV of PND^{SRC} consisting of the events that are pending in the source state and selected for delivery during that step;
- A set PRC that selects from INV^{SRC} one event for every component that has events waiting to be processed;
- A subset EXC of PRC consisting of the events that are actually executed (the others are discarded);
- A set PUB of the events that are published during that step together with a function that assigns a value to the parameters of each such event.

- The set INV^{TRG} of the events in the target state that have been invoked consists of the events in DLV (i.e. those that are delivered during the step) together with those already in INV^{SRC} that have not been selected by PRC to be processed;

- The set PND^{TRG} of the events that are pending at the target state consists of the events in PUB (i.e. those that are published during the step) together with the events in PND^{SRC} that have not been selected by DLV to be delivered.

That is, the set of events that are pending in wires is updated during each computation step by removing the events that the wire delivers during that step — DLV — and adding the events that each component publishes — PUB . We assume that all the events that are selected by DLV are actually delivered to the receiving component, i.e. each wire is reliable — see [4] for a model that considers unreliable wires.

At each step, components may select one of the events waiting to be processed; this is captured by the function PRC . The fact each component can only process one event at a time is justified by the assumption that the internal state of the components is not necessarily distributed and therefore no concurrent changes can be made to their states.

The set of events that are waiting to be processed by every component is updated in each step by removing the event that is processed and adding the events that are actually delivered to that component. Figure 7 is a graphical

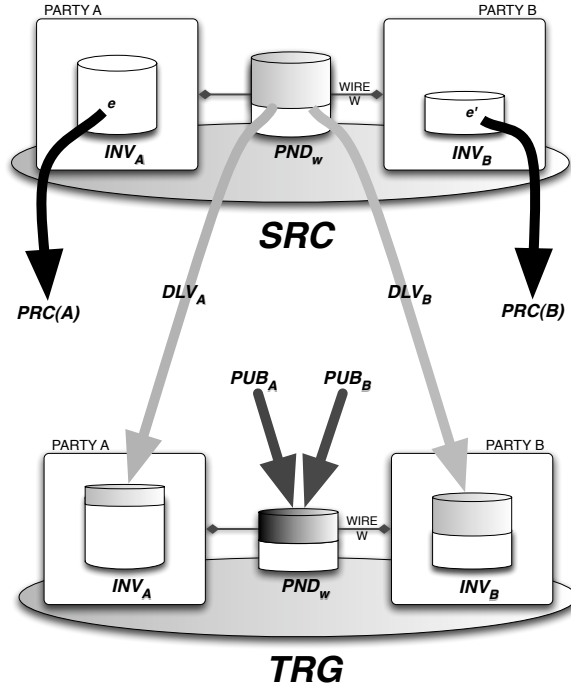


Fig. 7. Graphical representation of event flow from the point of view of a wire w between parties A and B .

representation of the flow of events that takes place during a computation step from the point of view of components A and B connected by a wire W .

5 The Modelling primitives of SRML

5.1 Behaviour specification languages

The entities involved in service and activity modules — component interfaces, requires-interfaces, provides-interfaces, uses-interfaces, serves-interfaces and wire-interfaces — can be defined in SRML independently of one another as design-time reusable resources. For that purpose, we have defined a number of different but related languages, which we present and illustrate in this section using fragments of our running example.

Signatures All the languages that we use have in common the declaration of the interactions (in the sense of Section 4.1) in which the corresponding entity can be involved — what we call a *signature*. These declarations are strictly local to the entity, i.e. we cannot rely on global names to establish interconnections between entities — that is the role of the wires. As an example, consider the component-interface MA , which we declared to be of type *MortgageAgent*. The corresponding signature is presented in Figure 8.

Interactions are classified according to the types defined in Section 4.1. For instance, *getProposal* is declared to be of type $\mathbf{r\&s}$, i.e. as being an asynchronous conversational interaction that is invoked by the co-party. This interaction has three parameters that carry data produced by the co-party at invocation time — the user profile, income and preferences for the mortgage. Such parameters are declared under the symbol \triangleleft . Parameters that are used by the mortgage agent for sending the reply are declared under the symbol \boxtimes — in the case at hand, the details of mortgage proposal and the cost of the mortgage-brokerage service for taking out the loan if the customer accepts the proposal.

The co-party of the mortgage agent in this interaction is not named (the same applies to all other interactions, as discussed in Section 4.1). This makes it possible to specify the behaviour that can be assumed of the mortgage agent at the interface, independently of the way it is instantiated within any given system.

The signature of *MortgageAgent* includes six additional interactions, all of which are self-initiated. While *askProposal*, *getInsurance*, *openAccount* and *signoutLoan* are conversational and asynchronous (i.e. of type $\mathbf{s\&r}$ or \mathbf{snd}), the interactions *getLenders* and *regContract* are synchronous. In the case of *getLenders*, the mortgage agent has to synchronise with the co-party to obtain data (the identification of the lenders that meet the user preferences for the mortgage) while, in the case of *regContract*, the party requests the co-party to perform an operation (register a loan contract) and blocks until the operation is completed.

Business roles In SRML, interfaces of service components are typed by *business roles*. A business role is specified by defining the way in which the interactions declared in the signature are orchestrated. For that purpose, we offer a textual declarative language based on states and transitions that is general enough to support languages and notations that are typically used for orchestrating workflows such as BPEL and UML statecharts.

In a typical business role, a set of variables provides an abstract view of the state of the component and a set of transitions models the activities performed by the component, including the way it interacts with its co-parties. For instance, the local state of a mortgage agent is defined as presented in Figure 9.

```

INTERACTIONS
  r&s getProposal
    Ⓐ idData:usrdata,
      income:moneyvalue,
      preferences:prefdata,
    ☒ proposal:mortgageproposal
      cost:moneyvalue
  s&r askProposal
    Ⓐ idData:usrdata,
      income:moneyvalue,
    ☒ proposal:mortgageproposal
      loanData:loandata,
      accountIncluded:bool,
      insuranceRequired:bool
  s&r getInsurance
    Ⓐ idData:usrdata,
      loanData:loandata,
    ☒ insuranceData:insurancedata
  s&r openAccount
    Ⓐ idData:usrdata,
      loanData:loandata,
    ☒ accountData:accountdata
  s&r signOutLoan
    Ⓐ insuranceData:insurancedata,
      accountData:accountdata,
    ☒ contract:loancontract
  snd confirmation
    Ⓐ contract:loancontract
  ask getLenders(prefdata):setids
  tll regContract(loandata,loancontract)

```

Fig. 8. The signature of *MortgageAgent*.

```

local  s:[INITIAL, WAIT_PROPOSAL, WAIT_DECISION,
          PROPOSAL_ACCEPTED, SIGNING, FINAL]
        lenders:setids
        needAccount, needInsurance:bool
        insuranceData:insurancedata, accountData:accountdata

```

Fig. 9. Local state of the *MortgageAgent*.

Typically, we use a variable (s in our example) to model control flow, including the way the component reacts to triggers. The other state variables are used for storing data that is needed at different stages of the orchestration.

Each transition has an optional name and a number of possible features. See Figure 10 for an example.

- A trigger is either the processing of an event, like in the example above, or a state condition. The former means that the transition is triggered when the component processes the event, and the latter when the condition changes from false to true.
- A guard is a condition that identifies the states in which the transition can take place — in *GetClientRequest*, the state *INITIAL*. If the trigger is an event and the guard is false, the event is processed but not executed (it is discarded).
- A sentence specifies the effects of the transition in the local state. Given a state variable var , we use var' to denote the value that var takes after the transition. In the case illustrated in Figure 10, we change the value of s and store the identification of the lenders that match the users-preferences. This data is obtained from a co-party through the synchronous interaction *getLenders*. As already mentioned, this co-party is not identified in the business role: we will see that, because of the way components are wired, the co-party in this interaction within the module GETMORTGAGE is *RE* of type *Registry* — the interface of a persistent component.

Another sentence specifies the events that are published during the transition, including the values taken by their parameters. In this sentence, we use variables and primed variables as in the ‘effects’-section. In the example, if there is at least one lender that matches the user-preferences, the interaction *askProposal* is initiated in order to get a mortgage proposal from a lender. Once again, the corresponding co-party is not named: we will see that, within the module GETMORTGAGE, this is an external service provided by a bank or building society that needs to be discovered and bound to the mortgage agent. If no lenders are found that match the user-preferences, a negative reply to *getProposal* is published.

```

transition GetClientRequest
  triggeredBy getProposal⊙
  guardedBy s=INITIAL
  effects lenders'=getLenders(prefdata)
    ∧ ¬empty(lenders') ⊃ s'=WAIT_PROPOSAL
    ∧ empty(lenders') ⊃ s'=FINAL
  sends ¬empty(lenders') ⊃ askProposal⊙
    ∧ askProposal.idData=getProposal.idData
    ∧ askProposal.income=getProposal.income
    ∧ empty(lenders') ⊃ getProposal⊠
    ∧ getProposal.Reply=false

```

Fig. 10. Transition *GetClientRequest*.

Another example of a transition is *GetLenderProposal* presented in Figure 11. In this case, the transition is triggered by the processing of the reply to *askProposal* and the effect is to send a reply to *getProposal* (the parameter *Reply* of *askProposal* and the proposal received in proposal are both transmitted by the reply-event). The transition also defines the cost of the mortgage-brokerage service for taking out the loan if the customer accepts the proposal.

Specifications may also declare configuration variables, which are discussed in Section 5.2. These variables are instantiated at run time, when a new session of the service starts, possibly as a result of the negotiation process involved in the discovery of the service. In the case of *MortgageAgent*, we declare the configuration variable CHARGE that determines an additional charge over the base price of the mortgage-brokerage service. In Section 5.2 we will see that, in the module GETMORTGAGE, this extra-charge relates to the period of validity of the loan proposal offered by the service, which is also subject to negotiation.

Notice that, through business roles, SRML offers a very flexible way for modelling control flow because transitions are decoupled from interactions and changes to state variables, which offers a declarative style of defining orchestrations. For instance, the transition *TimeoutProposal* defined below is triggered once the reply to *getProposal* expires; in this situation, the component informs the lender that the proposal was not accepted and moves to the final state.

Other aspects of this declarative style include the possibility of leaving certain aspects under-specified that can be refined at later stages of the development process. This is why the various aspects of a transition are specified as sentences using a logical notation.

More traditional (control-oriented) notations can be used instead for defining orchestrations. In Figure 13 we show how part of the orchestration of *MortgageAgent* can be defined using a UML statechart. Because statecharts focus only

```

transition GetLenderProposal
  triggeredBy askProposal⊠
  guardedBy s=WAIT_PROPOSAL
  effects needAccount'=askProposal.accountIncluded
    ∧ needInsurance'=askProposal.insuranceRequired
    ∧ askProposal.Reply ⊃ s'=WAIT_DECISION
    ∧ ¬askProposal.Reply ⊃ s'=FINAL
  sends getProposal⊠
    ∧ getProposal.Reply=askProposal.Reply
    ∧ getProposal.proposal=askProposal.proposal
    ∧ getProposal.cost=(CHARGE/100+1)*750

```

Fig. 11. Transition *GetLenderProposal*.

```

transition TimeoutProposal
  triggeredBy now>getProposal.UseBy
  guardedBy s=WAIT_DECISION
  effects s'=FINAL
  sends askProposal*

```

Fig. 12. Transition *TimeOutProposal*.

on control flow, we would need to provide a separate specification for the data flow. In [15], we have also shown how BPEL can be encoded in our language.

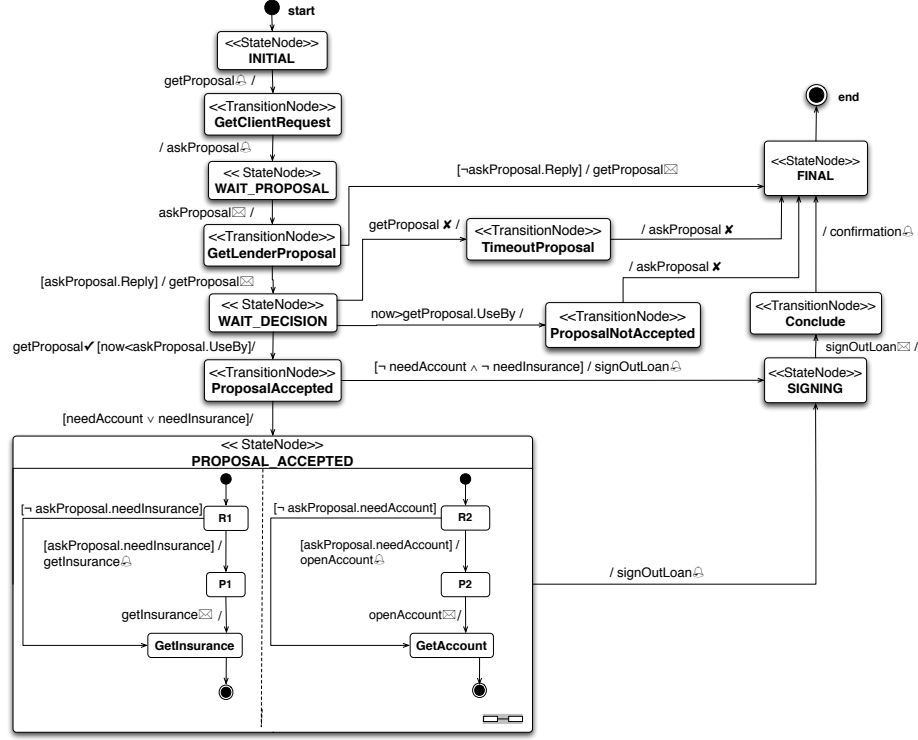


Fig. 13. Using UML statecharts for defining orchestrations in business roles.

Business protocols In SRML, a module may declare a number of requires-interfaces, each of which provides an abstraction (type) for a service that will have to be procured from external providers, if and when needed — what, in SCA, corresponds to an “External Service”. In the case of a service module, a provides-interface is also declared for describing the service that is offered by the module, corresponding to what in SCA is called an “Entry Point”.

Both types of external interfaces are typed with what we call business protocols, or just protocols if it is clear from the context what kind of protocols we are addressing. Like business roles, protocols include a signature. The difference is that, instead of an orchestration, we provide a set of properties. In the case of a requires-interface, these are the properties required of the external service that needs to be procured. In the case of a provides-interface, we specify the properties offered by the service orchestrated by the module.

In the case of business protocols used for specifying the required services, we declare the interactions in which the external entity (to be procured) must be able to be involved as a (co-)party and we specify the protocol that it has to adhere to. For instance, the service GETMORTGAGE expects the behaviour from a lender described in Figure 14.

```

BUSINESS PROTOCOL Lender is


---


INTERACTIONS
  r&s requestMortgage
    Ⓐ idData:usrdata,
      income:moneyvalue,
    ☒ proposal:mortgageproposal
      loanData:loandata,
      accountIncluded:bool,
      insuranceRequired:bool

  r&s requestSignOut
    Ⓐ insuranceData:insurancedata,
      accountData:accountdata,
    ☒ contract:loancontract

BEHAVIOUR
  initiallyEnabled requestMortgageⒶ?
  requestMortgage✓? enables requestSignOutⒶ?

```

Fig. 14. The specification of business protocol *Lender*.

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

- In the initial state, the lender is ready to engage in *requestMortgage*.
- After receiving the commitment to the mortgage proposal, the lender becomes ready to engage in *requestSignOut*.

The language in which these properties are expressed uses a set of patterns that capture commonly occurring requirements in the context of service-oriented interactions. In Section 7.1, we present their semantics in terms of formulas of the temporal logic UCTL [51]. Intuitively, they correspond to traces of the form depicted in Figure 15.

The intuitive semantics of these patterns is as follows:

- **initiallyEnabled** *e*: The event *e* is enabled (cannot be discarded) in the initial state and remains so until it is executed.
- *s* **after** *a*: the state condition *s* holds forever after the action condition *a* becomes true.
- *a* **enables** *e* **until** *b*: The event *e* cannot be executed before *a* holds and remains enabled after *a* becomes true until it is either executed or *b* becomes true (if ever).

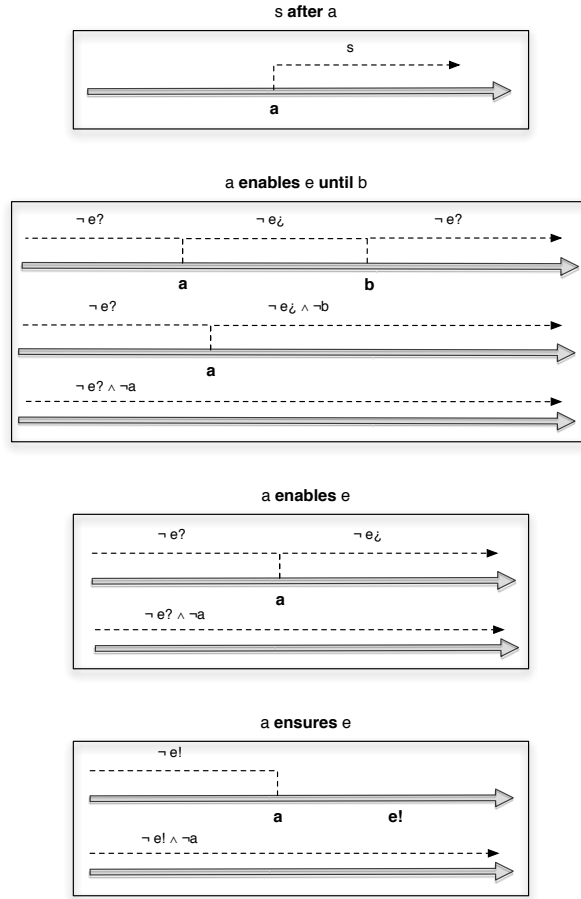


Fig. 15. The traces that correspond to the patterns.

- ***a enables e***: The event *e* cannot be executed before *a* holds and remains enabled after *a* becomes true until it is executed. It is easy to see that this pattern is equivalent to ***a enables e until false***.
- ***a ensures e***: The event *e* cannot be published before *a* holds, and is published sometime after *a* becomes true.

Business protocols are also used for modelling the behaviour that users can expect from a service. This subsumes what, in [8], are called external specifications:

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

For instance, the provides-interface of GETMORTGAGE is typed by the business protocol presented in Figure 16.

This business protocol specifies that the service offered by GETMORTGAGE relies on two asynchronous interactions — *getProposal* and *confirmation*. The properties offered by the service are:

- A request for *getProposal* is enabled when the service is activated.
- The service brokerage has a base price that can be subject to an extra charge, subject to negotiation.
- A confirmation carrying the loan contract will be issued upon receipt of the commit to *getProposal*.

Layer protocols A module in SRML may also declare one or more uses-interfaces. These provide abstractions of components corresponding to resource actors as discussed in Section 3.1 — the components with which the service needs to interact in order to ensure persistent effects.

Uses-interfaces are specified through what we call layer protocols. Like business protocols, layer protocols are defined by a signature and a set of properties. However, where the interactions used in business protocols are asynchronous, those declared in a layer protocol can be synchronous and blocking.

As an example, consider the specification of the layer protocol fulfilled by a registry as shown in Figure 17. It defines that a registry can be queried — through the interaction *getLenders* — about the registered lenders that meet given users preferences, and is able to register a new contract through the operation *registerContract*.

The properties of synchronous interactions are typically in the style of pre/post-condition specifications of methods.

```

BUSINESS PROTOCOL Customer is


---


INTERACTIONS
  r&s getProposal
    ⚡ idData:usrdata,
      income:moneyvalue,
      preferences:prefdata,
    ☒ proposal:mortgageproposal,
      cost:moneyvalue
  snd confirmation
    ⚡ contract:loancontract
SLA VARIABLES
  CHARGE:[0..100]
BEHAVIOUR
  initiallyEnabled getProposal⚡?
  getProposal.cost≤750*(CHARGE/100+1) after
    (getProposal☒! ^ getProposal.Reply)
  getProposal✓? ensures confirmation⚡!

```

Fig. 16. The specification of business protocol *Customer*.

Interaction protocols A module consists of a number of interfaces connected through wires. Wires are labelled by connectors that coordinate the interactions in which the parties are jointly involved. In SRML, we model the interaction protocols involved in these connectors as separate, reusable entities.

Just like business roles and protocols, an interaction protocol is specified in terms of a number of interactions. Because interaction protocols establish a relationship between two parties, the interactions in which they are involved are divided in two subsets called roles — A and B . The semantics of the protocol is provided through a collection of sentences — what we call interaction glue — that establish how the interactions are coordinated.

As an example, consider the protocol depicted in Figure 18, which is used in the wire that connects *MortgageAgent* and *Insurance*. This is a ‘straight’ protocol that connects directly two entities over two conversational interactions that have two \triangleleft -parameters and one \boxtimes -parameter. The property $S_1 \equiv R_1$ establishes that the events associated with each interaction are the same, e.g. that S_1 is the same as R_1 .

The names used in interaction protocols are generic to facilitate reuse. In fact, the specification itself is parameterised by the data sorts involved in the interactions. Parameterisation (which is also available for business roles and protocols) provides the means for defining families of specifications. The parameters are instantiated at design time when the specifications are used in the definition of a module. This can be seen at the end of this Section.

```
LAYER PROTOCOL Registry is


---


INTERACTIONS
  rp1 getLenders(prefdata):setids
  prf registerContract(loandata,loancontract)
BEHAVIOUR
```

Fig. 17. The specification of layer protocol *Registry*.

```
INTERACTION PROTOCOL Straight.I(d1,d2)O(d3) is


---


ROLE A
  s&r S1
     $\triangleleft$  i1:d1, i2:d2
     $\boxtimes$  o1:d3
ROLE B
  r&s R1
     $\triangleleft$  i1:d1, i2:d2
     $\boxtimes$  o1:d3
COORDINATION
  S1 = R1
  S1.i1=R1.i1
  S1.i2=R1.i2
  S1.o1=R1.o1
```

Fig. 18. The specification of an interaction protocol.

Two other families of straight protocols are presented below. These families define the connection of two synchronous interactions with two parameters; in the first protocol, the interaction involves a return value. The first interaction protocol establishes that the values returned by the synchronous interaction are the same, while the second protocol synchronises the two operations without any conversion of data.

Interaction protocols are first-class objects that can be (re)used to assign properties to wires, which reflect constraints on the underlying run-time environment. These may concern data transmission, synchronous/asynchronous connectivity, distribution, and other non-functional properties such as security. In such cases, the specifications are not as simple as those of straight protocols.

Connectors After having chosen the protocols that coordinate the interactions between two parties, we use them as the ‘glue’ (in the sense of [48]) of the connectors that label the wires that link the corresponding parties. In a connector, the interaction protocol is bound to the parties via ‘attachments’: these are mappings from the roles to the signatures of the parties identifying which interactions of the parties perform which roles in the protocol. The use of attachments allows us to separate the definition of the interaction protocols from their use in the wires, which promotes reuse: typically, one defines a connector by choosing from a repository of (types of) protocols that have proved to be useful in other situations.

Summarising, connectors are triples $\langle \mu_A, P, \mu_B \rangle$ where:

- P is an interaction protocol. We use $roleA_P$ and $roleB_P$ to designate its roles and $glue_P$ for the role.
- μ_A and μ_B are attachments that connect the roles of the protocol to the signatures of the entities (business roles, business protocols or layer protocols) being interconnected.

```

INTERACTION PROTOCOL Straight.A( $d_1, d_2$ )R( $d_3$ ) is


---


  ROLE A
    ask  $S_1(d_1, d_2) : d_3$ 
  ROLE B
    rpl  $R_1(d_1, d_2) : d_3$ 
  COORDINATION
     $S_1(d_1, d_2) = R_1(d_1, d_2)$ 

INTERACTION PROTOCOL Straight.T( $d_1, d_2$ ) is


---


  ROLE A
    tll  $S_1(d_1, d_2)$ 
  ROLE B
    prf  $R_1(d_1, d_2)$ 
  COORDINATION
     $S_1(d_1, d_2) \equiv R_1(d_1, d_2)$ 

```

Fig. 19. Another two specifications of interaction protocols.

For instance, both *Straight.A(prefdata)R(setids)* and *Straight.T(loandata, loancontract)* are used in the wire *ME* to connect different interactions between *MortgageAgent* and *Registry* as depicted in Figure 20.

Each row describes one connector. The first two columns define the attachment between *roleA* of the interaction protocol (specified in the middle column) and the signature of *MortgageAgent*. In the same way, the last two columns define the attachment between *roleB* of the interaction protocol and the signature of *Registry*.

We use the same notation for specifying the wires that connect module components to requires-interfaces. However, the specification of these wires is subject to an additional correctness condition that restricts the signature of the requires-interfaces to the interaction used in the corresponding wires. This is to ensure that all the interactions of the services that are bound to the module through the requires-interface have a corresponding co-party.

For instance, the only wire that connects *LE* in GETMORTGAGE is *ML* (with *MA*). Its specification is presented in Figure 21. The correctness condition is satisfied because the signature of *Lender* is isomorphic to the sum of the interactions of the roles connected to it, i.e. all the interactions of *Lender* are mapped to a port.

The specification of the wires that connect module components to the provides-interface of the module uses a slightly different syntax. This is because what we need to declare is the set of interactions that the components make available to the customer of the service, and the protocols through which the corresponding

MA MortgageAgent	c_4	BE	d_4	RE Registry
ask getLenders	S_1	Straight. A(prefdata)R(setids)	R_1	rpl getLenders
tll regContract	S_1	Straight. T(loandata, loancontract)	R_1	prf registerContract

Fig. 20. The specification of the connectors involved in wire *ME*.

MA MortgageAgent	c_1	CL	d_1	LE Lender
s&r askProposal Ⓐ idData income ☒ proposal loanData accountIncluded insuranceRequired	S_1 i_1 i_2 o_1 o_2 o_3 o_4	Straight. I(usrdData, moneyvalue) O(mortgageproposal, loandata, bool,bool)	R_1 i_1 i_2 o_1 o_2 o_3 o_4	r&s requestMortgage Ⓐ idData income ☒ proposal loanData accountIncluded insuranceRequired
r&s signOutLoan Ⓐ insuranceData accountData ☒ contract	S_1 i_1 i_2 o_1	Straight I(insurancedata, accountdata) O(loancontract)	R_1 i_1 i_2 o_1	s&r requestSignOut Ⓐ insuranceData accountData ☒ contract

Fig. 21. The specification of the connectors involved in wire *ML*.

events are transmitted. In this sense, we do not model the customer proper, which in SRML is reflected by omitting the corresponding column of the table that defines the wire.

For instance, the wire *CM* that interconnects *Customer* and *MortgageAgent* in GETMORTGAGE is specified as presented in Figure 22. In this case, each row also describes one connector whose interaction protocol is specified in the second column. The difference is that the entities that will be connected to the *roleA* of their interaction protocols are unknown (these will belong to the services that will bind to GETMORTGAGE). As before, the last two columns define the attachment between *roleB* of the interaction protocol and the signature of *MortgageAgent*.

5.2 Configuration policies

Whereas business roles, business protocols, layer protocols and interaction protocols deal with functional aspects of the behaviour of a (complex) service or activity, configuration policies address aspects that relate to processes of discovery, selection and instantiation of services. In SRML, we distinguish between internal and external configuration policies. The former concern aspects related with service instantiation such as the initialization of service components and the triggering of the discovery of required services. The latter address aspects related with the selection of partner services and negotiation of contracts.

Internal configuration policy The internal configuration policy of a service module concerns the triggering of the discovery and selection process associated with its requires-interfaces, and the instantiation of its component and wire interfaces.

A trigger is usually associated with the occurrence of one or more events and additional conditions on the state of the components in which the events occur. For instance, GETMORTGAGE defines that the lender has to be discovered as soon as *getProposal*_Δ is executed (by the workflow). There is a *default* trigger condition: the publication of the initiation event of the first interaction connected to the requires-interface. In our example, this is the case of the bank and insurance external services.

c ₁	CB	d ₁	MA MortgageAgent
S ₁ i ₁ i ₂ i ₃ o ₁ o ₂	Straight. I(usrdata, moneyvalue,prefdata) O(mortgageproposal, moneyvalue)	R ₁ i ₁ i ₂ i ₃ o ₁ o ₂	r&s getProposal Δ idData income preferences ☒ proposal cost
R ₁ i ₁	Straight. O(loancontract)	S ₁ i ₁	snd confirmation Δ contract

Fig. 22. The specification of the connectors involved in wire *CM*.

In a module, each service component has an associated initialisation condition, which is guaranteed to hold when the component is instantiated, and a termination condition, which determines when the component stops executing and interacting with the rest of the components (in which case it can be removed from the state configuration to which it belongs). Typically, both conditions relate to the state variables of the component, but they can also include the publication of given events. For instance, in the case of *MortgageAgent*, these conditions are defined only in terms of the local variable *s* (see Figure 24).

Notice that these conditions can be underspecified, leaving room for further refinement. For instance, we may force the termination of the component after a certain date without specifying exactly when.

External policies The external policy concerns the way the module relates to external parties: it declares the set of variables that can be used for negotiation and establishing a service level agreement (SLA), and a set of constraints that have to be taken into account during discovery and selection.

SLA variables include all the configuration variables declared in the specifications (except in the provides-interface). For instance, in GETMORTGAGE, *MortgageAgent* declares the configuration variable CHARGE. These variables are local to the interfaces to which they are attached and instantiated when the corresponding component is created. Because constraints apply to the module as a whole, we refer to these variables by preceding them with the name of the entity to which they belong. Hence, in GETMORTGAGE, we refer to *MA.CHARGE*.

SRML also provides a set of standard configuration variables — availability, response time, message reliability, service identification, inter alia. Some of them, e.g. response time, are associated with requires or provides-interfaces, and other, e.g. message reliability, apply to the wires.

The standard configuration variables used in GETMORTGAGE are:

- *interaction*^{*}, for every interaction of type *r*&*s*; its value is the length of time the reply is valid after *interaction* is issued.

```
LE: Lender
    intLE⌚trigger: getproposal⌚?
BA: Bank
    intBA⌚trigger: default
IN: Insurance
    intIN⌚trigger: default
```

Fig. 23. Trigger conditions in GETMORTGAGE.

```
MA: MortgageAgent
    intMA⌚init: s=INITIAL
    intMA⌚term: s=FINAL
```

Fig. 24. Initialization and termination conditions in GETMORTGAGE.

- *wire.Delay*, for every wire; it defines the maximum delivery delay for events sent over that wire.
- *ServiceId*, for every external-interface; it represents the identification of the service that is bound to that interface (for instance, a URI).

Notice that although these variables are standard they need to be declared in a module if the designer wants them to be involved in the service discovery negotiation process. For instance, their declaration in GETMORTGAGE is presented in Figure 25.

The approach that we adopt in SRML for SLA negotiation is based on the constraint satisfaction and optimization framework presented in [12] in which constraint systems are defined in terms of c-semirings. As explained therein, this framework is quite general and allows us to work with constraints of different kinds — both hard and ‘soft’, the latter in many grades (fuzzy, weighted, and so on). The c-semiring approach also supports selection based on a characterisation of ‘best solution’ supported by multi-dimensional criteria, e.g. minimizing the cost of a resource while maximizing the work it supports.

In this framework:

- A c-semiring is a semiring $\langle A, +, \times, 0, 1 \rangle$ in which A represents a space of degrees of satisfaction, e.g. the set $\{0, 1\}$ for yes/no or the interval $[0, 1]$ for intermediate degrees of satisfaction. The operations \times and $+$ are used for composition and choice, respectively. Composition is commutative, choice is idempotent and 1 is an absorbing element (i.e. there is no better choice than 1). That is, a c-semiring is an algebra of degrees of satisfaction. Notice that every c-semiring S induces a partial order \leq_S (of satisfaction) over A as follows: $a \leq_S b$ iff $a + b = b$. That is, b is better than a iff the choice between a and b is b .
- A constraint system is a triple $\langle S, D, V \rangle$ where S is a c-semiring, V is a totally ordered set (of configuration variables), and D is a finite set (domain of possible elements taken by the variables).
- A constraint consists of a selected subset con of variables and a mapping $def : D^{con} \rightarrow S$ that assigns a degree of satisfaction to each tuple of values taken by the variables involved in the constraint.

The external configuration policy of a module involves a constraint system based on a fixed c-semiring and a set of constraints over this constraint system. Because we want to handle constraints that involve different degrees of satisfaction, it makes sense that we work with the c-semiring $\langle [0..1], max, min, 0, 1 \rangle$ of soft fuzzy constraints [12]. In this c-semiring, the preference level is between 0 (worst) and 1 (best).

SLA VARIABLES

```
MA.CHARGE, MA.getProposal∞,
LE.ServiceId, LE.Cost, LE.requestMortgage∞
```

Fig. 25. Declaration of SLA variables in GETMORTGAGE.

For instance, the external configuration policy of GETMORTGAGE includes the following constraints:

$$C_1 : \{MA.CHARGE, MA.getProposal\bullet^{\otimes}\},$$

$$def(c, t) = \begin{cases} 1 & \text{if } t \leq 10 \cdot c \\ 1 + 2 \cdot c - 0.2 \cdot t & \text{if } 10 \cdot c < t \leq 5 + 10 \cdot c \end{cases}$$

That is, the more CHARGE is applied to the base price of the brokerage service the longer is the interval during which the proposal is valid.

$$C_2 : \{LE.ServiceId\}, \quad def(s) = \begin{cases} 1 & \text{if } s \in MA.lenders \\ 0 & \text{otherwise} \end{cases}$$

That is, the choice of the lender is constrained by the service identifier, which must belong to the set *MA.lenders* (recall that, according to the orchestration of *MortgageAgent*, this set contains the identification of the services provided by trusted lenders that were found to be appropriate for the request at hand).

$$C_3 : \{MA.getProposal\bullet^{\otimes}, LE.requestMortgage\bullet^{\otimes}\},$$

$$def(t_1, t_2) = \begin{cases} 1 & \text{if } t_2 > t_1 + CM.Delay + ML.Delay \\ 0 & \text{otherwise} \end{cases}$$

That is, the choice of the lender is also constrained by the period of validity associated with its loan proposals. This period must be greater than the sum of the validity period offered by the brokerage service to its clients and the possible delays that may affect the transmission through the wires involved (notice that *CM.Delay* and *ML.Delay* are not declared as SLA variables and, hence, they are used like constants).

$$C_4 : \{LE.COST, LE.requestMortgage\bullet^{\otimes}\}, \quad def(c, t) = \begin{cases} \frac{1}{c} + \frac{t}{100} & \text{if } c < 500 \\ 0 & \text{otherwise} \end{cases}$$

That is, the cost to be paid by the brokerage service to the lender must be less than 500, and the preference between lenders charging the same value will take into account the validity period of the loan proposals.

The value of SLA variables is negotiated during service discovery/binding. Details on negotiation of constraints and SLAs are further discussed in Section 6.3.

5.3 Module declaration

SRML makes available a textual language for defining modules, which involves the specification of the module external interfaces, service components, wires

and policies, as discussed in the previous sections. The full definition of GETMORTGAGE can be seen in Appendix B.

In the case of a service module, we also have to map the interactions and SLA variables of the provides-interface to corresponding interactions and variables of the entities that provide the service. This is because the business protocol that labels the provides-interface represents the service that is offered by the module (behavioural properties and negotiable SLA variables), not the activity to which the service will be bound. In the case of GETMORTGAGE, only *MA* is connected to *CR*, so the mapping is actually an identity. This is specified as presented in Figure 26.

6 The Configuration-Management Model

6.1 Layered state configurations of global computers

As already mentioned, we take SOC to be about applications that can bind to other applications discovered at run time in a universe of resources that is not fixed a priori. As a result, there is no structure or ‘architecture’ that one can fix at design-time for an application; rather, there is an underlying notion of configuration of a global computer that keeps being redefined as applications execute and get bound to other applications that offer required services. As is often the case (e.g. [48]), by ‘configuration’ we mean a graph of components (applications deployed over a given execution platform) linked through wires (e.g. interconnections between components over a given communication network) in a given state of execution. Typically, wires deal with the heterogeneity of partners involved in the provision of the service, performing data (or, more, generally, semantic) integration. See Figure 27 for an example, over which we will later recognise three business activities (instances).

Summarising, a *state configuration* \mathcal{F} consists of:

PROVIDES	
CR: Customer	
CR Customer	MA MortgageAgent
r&s getProposal ⌚ idData income preferences ✉ proposal cost	r&s getProposal ⌚ idData income preferences ✉ proposal cost
snd confirmation ⌚ contract	snd confirmation ⌚ contract
SLA VARIABLES CHARGE	SLA VARIABLES CHARGE

Fig. 26. Specification of the mapping between *CR* and *MA* in GETMORTGAGE.

- A simple graph \mathcal{G} , i.e. a set $nodes(\mathcal{F})$ and a set $edges(\mathcal{F})$; each edge e is associated with a (unordered) pair $n \leftrightarrow m$ of nodes. We take $nodes(\mathcal{F}) \subseteq \mathbf{COMP}$ (i.e. nodes are components) and $edges(\mathcal{F}) \subseteq \mathbf{WIRE}$ (i.e. edges are wires).
- A (configuration) state \mathcal{S} as defined in Section 4.3.

Every state configuration $\langle \mathcal{G}, \mathcal{S} \rangle$ can change because either the state \mathcal{S} or the graph \mathcal{G} changes. Changes to the state result from computations executed by components and the coordination activities performed by the wires that connect them as defined in 4.3. However, the essence of SOC as we see it is not captured at the level of state changes (which is basically a distributed view of computation), but at the level of the changes that operate on configuration graphs: in SOC, changes to the underlying graph of components and wires occur at run time when a component performs an action that triggers the discovery and binding of a service.

An important aspect of our model is the fact that we view SOC as providing an architectural layer that interacts with two other layers (see Figure 28). This can be noticed in Figure 27 where shadows are used for indicating that certain components reside in different layers: *AliceRegUI*, *BobEstateUI* and *CarolEstateUI* (three user interfaces) in the top layer, and *MyRegistry* (a database) in the bottom layer. Layers are architectural abstractions that reflect different levels of organisation and change, i.e. one looks at a configuration as a (flat) graph as indicated above but, in order to understand how such configurations evolve, it is useful to distinguish different layers.

In our model, the bottom layer consists of components that are persistent as far as the service layer is concerned, i.e. those that in Section 3 we identified as resource-actors. More precisely, when a new session of a service starts (e.g. a mortgage broker starts putting together a proposal on behalf of a client),

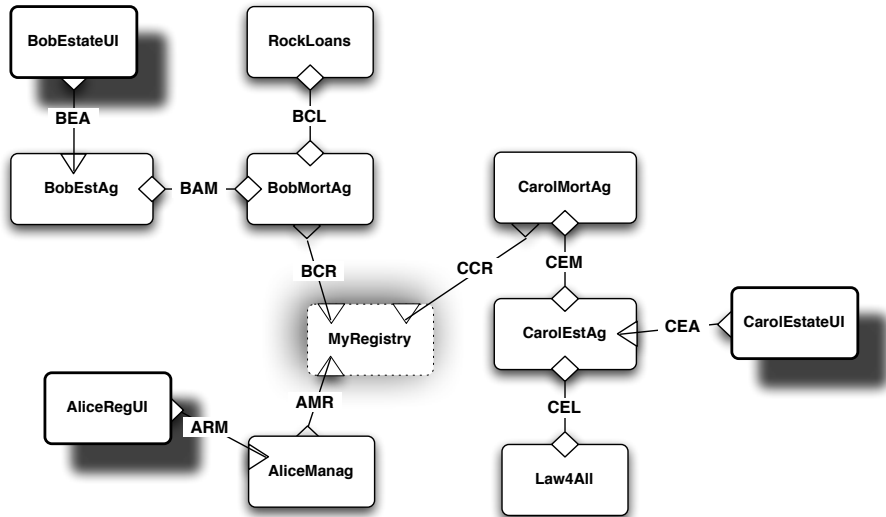


Fig. 27. The graph of a state configuration with 11 components and 10 wires.

the components of the bottom layer are assumed to be available so that, as the service executes, they can be used as (shared) ‘servers’ — for instance the registry, which is shared by all sessions of the mortgage broker, or a currency converter. In particular, the bottom layer can be used for making persistent the effects of services as they execute.

The components that execute in the service layer are created when the session of the corresponding service starts, i.e. as fresh instances that last only for the duration of the session — for instance, the workflow that orchestrates the mortgage-brokerage service for a particular client. In component-based development (CBD) one often says that the bottom layer provides ‘services’ to the layer above. As we see it in this paper, an important difference between CBD and SOC is precisely in the way such services are procured, which in the case of SOC involves identifying (possibly new) providers and negotiating terms and conditions for each new instance of the activity, e.g. for each new user of a travel agent. SOA middleware supports this service layer by providing the infrastructure for the discovery and negotiation processes to be executed without having to be explicitly programmed as (part of) components.

The top layer is the one responsible for launching business activities in the service layer. The user of a given activity — identified through a user-actor as discussed in Section 3 — resides in the top layer; it can be an interface for human-computer interaction, a software component, or an external system (e.g. a control device equipped with sensors). When the user launches an activity, a component is created in the service layer that starts executing a workflow that may involve the orchestration of services that will be discovered and bound to the workflow at run time.

6.2 Business activities and configurations

In our model, state configurations change as a result of the execution of business processes. More precisely, changes to the configuration graph result from the fact that the discovery of a service is triggered and, as a consequence, new components are added and bound to existing ones (and, possibly, other compo-

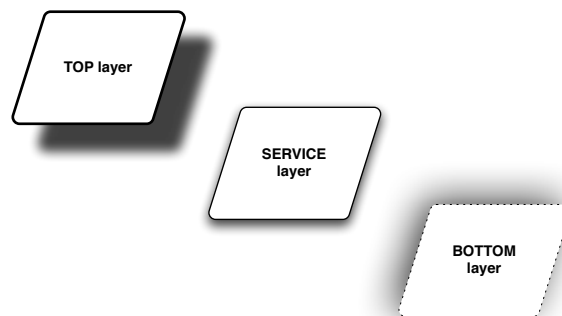


Fig. 28. A 3-layered architecture for configurations.

nents and wires disappear because they finished executing their computations). The information about the triggers and the constraints that apply to service discovery and binding are not coded in the components themselves: they are properties of the ‘business activities’ that are active and determine how the configuration evolves. Thus, in order to capture the dynamic aspects of SOC, we need to look beyond the information available in a state. In our approach, we achieve this by making configurations ‘business reflective’, i.e. by labelling the sub-configurations that correspond to instances of business activities by the corresponding activity module.

For instance, we should be able to recognise an activity in Figure 27 whose sub-configuration is as depicted in Figure 29. Intuitively, it corresponds to an instance of `UPDATEREGISTRY`. In order to formalise this notion of typed sub-configuration, we start by providing a formal definition of activity modules. We denote by **BROL** the set of business roles (see 5.1.2), by **BUSP** the set of business protocols (see 5.1.3), by **LAYP** the set of layer protocols (see 5.1.4), and by **CNCT** the set of connectors (see 5.1).

An *activity module* M consist of:

- A graph $graph(M)$.
- A distinguished subset of nodes $requires(M) \subseteq nodes(M)$.
- A distinguished subset of nodes $uses(M) \subseteq nodes(M)$.
- A node $serves(M) \in nodes(M)$ distinct from $requires(M)$ and $uses(M)$.
- A labelling function $label_M$ such that
 - $label_M(n) \in \mathbf{BROL}$ if $n \in components(M)$, where by $components(M)$ we denote the set of $nodes(M)$ that are not $serves(M)$ nor in $requires(M)$ or $uses(M)$.
 - $label_M(n) \in \mathbf{BUSP}$ if $n \in requires(M)$
 - $label_M(n) \in \mathbf{LAYP}$ if $n \in serves(M) \cup uses(M)$
 - $label_M(e : n \leftrightarrow m) \in \mathbf{CNCT}$.
- An internal configuration policy.
- An external configuration policy.

We denote by $body(M)$ the (full) sub-graph of $graph(M)$ that forgets the nodes in $requires(M)$ and the edges that connect them to the rest of the graph.

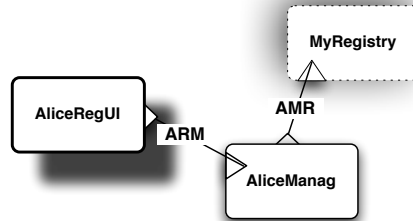


Fig. 29. The sub-configuration corresponding to an instance of `UPDATEREGISTRY`.

We can now formalise the typing of state configurations with activity modules that we discussed around Figure 29, which accounts for the coarser business dimension that is overlaid by services on global computers. That is, we define what corresponds to a state configuration of a service overlay computer, which we call a business configuration. We consider a space \mathcal{A} of business activities to be given, which can be seen to consist of reference numbers (or some other kind of identifier) such as the ones that organisations automatically assign when a service request arrives.

A *business configuration* consists of:

- A state configuration \mathcal{F} .
- A partial mapping \mathcal{B} that assigns an activity module $\mathcal{B}(a)$ to each activity $a \in \mathcal{A}$ — the workflow being executed by a in \mathcal{F} . We say that the activities in the domain of this mapping are those that are active in that state.
- A mapping \mathcal{C} that assigns an homomorphism $\mathcal{C}(a)$ of graphs $\text{body}(\mathcal{B}(a)) \rightarrow \mathcal{F}$ to every activity $a \in \mathcal{F}$ that is active in \mathcal{F} . We denote by $\mathcal{F}(a)$ the image of $\mathcal{C}(a)$ — the sub-configuration of \mathcal{F} that corresponds to the activity a .

A homomorphism of graphs is just a mapping of nodes to nodes and edges to edges that preserves the end-points of the edges. Therefore, the homomorphism \mathcal{C} of a business configuration $\langle \mathcal{F}, \mathcal{B}, \mathcal{C} \rangle$ types the nodes (components) of $\mathcal{F}(a)$ with business roles or layer protocols — i.e. $\mathcal{C}(a)(n) : \text{label}_{\mathcal{B}(a)}(n)$ for every node n — and the edges (wires) with connectors — i.e. $\mathcal{C}(a)(e) : \text{label}_{\mathcal{B}(a)}(e)$ for every edge e of the body of the activity. In other words, the homomorphism binds the components and wires of the state configuration to the business elements (interfaces labelled with business roles, layer protocols and connectors) that they fulfil in the activity.

In the example discussed above, we have an activity — that we call *Alice* — such that $\mathcal{B}(\text{Alice})$ is `UPDATEREGISTRY` (as in Figure 3), $\mathcal{F}(\text{Alice})$ is the sub-configuration in Figure 29, and \mathcal{C} maps *RM* to *AliceRegUI*, *MC* to *AliceManag*, *RE* to *MyRegistry*, *MR* to *AMR*, and *RM* to *ARM*.

The fact that the homomorphism is defined over the body of the activity module means that business protocols are not used for typing components of the state configuration. Indeed, as discussed above, the purpose of the requires-interfaces is for identifying dependencies that the activity has, in that state, on external services. In particular, this makes requires-interfaces different from uses-interfaces as the latter are indeed mapped through the homomorphism to a component of the state configuration.

In a sense, the homomorphism makes state configurations reflective in the sense of [27] as it adds meta (business) information to the state configuration. This information is used for deciding how the configuration will evolve (namely, how it will react to events that trigger the discovery process). Indeed, reflection has been advocated as a means of making systems adaptable through reconfiguration, which is similar to the mechanisms through which activities evolve in our model.

6.3 Run-time discovery and binding

In order to illustrate how a business configuration evolves through service discovery and binding, we are going to consider another business activity type that supports the purchase of a house. The corresponding module is depicted in Figure 30.

That is, the orchestration of the purchase of a house is performed by a component EA of type (business role) EstateAgent, which may need to discover and bind to a mortgage dealer *MO* and a lawyer *LA*.

Consider the configuration depicted in Figure 31, and the business configuration that consists of *Alice* (as defined in Section 6.2) and of the activity *Bob* typed by HOUSEBUYING, which is mapped to the configuration by the homomorphism that associates *GH* with *BobEstateUI*, *EA* with *BobEstateAG* and *HE* with *BEA*. Assume that, in the current state, *intMO* trigger holds, i.e. that the execution of the workflow associated with *EA* requires the discovery of a mortgage dealer. Let us consider what is necessary for GETMORTGAGE to be selected and bound to HOUSEBUYING as a result of the trigger (see Figure 32). In our setting, this process involves three steps, outlined as follows:

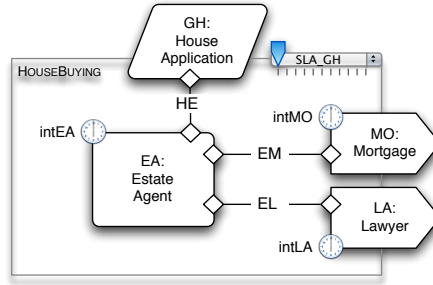


Fig. 30. The HOUSEBUYING activity module.

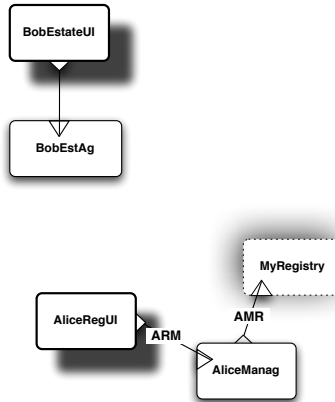


Fig. 31. A configuration.

- **Discovery.** For GETMORTGAGE to be discovered, it is necessary that the properties of its provides-interface *Customer* entail the properties of the requires-interface *Mortgage*, and that the properties of the interaction protocol of *CC* entail those of *EM*.
- **Ranking.** If it is discovered, GETMORTGAGE is ranked among all services that are discovered by calculating the most favourable service-level agreement that can be achieved — the contract that will be established between the two parties if GETMORTGAGE is selected. This calculation uses a notion of satisfaction that takes into account the preferences of the activity HOUSEBUYING and the service GETMORTGAGE.
- **Selection.** Finally, GETMORTGAGE can be selected if it is one of the services that maximises the level of satisfaction offered by the corresponding contract.

These steps are formalised in [33]. If GETMORTGAGE is selected then it is unified with HOUSEBUYING, giving rise to another activity module. As depicted in Figure 33, the resulting activity module is obtained by replacing the requires-interface and corresponding wire of HOUSEBUYING by those that connect the provides-interface of GETMORTGAGE to its body.

At the level of the configuration, we add the new instances of the components of GETMORTGAGE and corresponding wires, making sure that instances of the uses-interfaces are components of the bottom layer (already present in the configuration). This can be witnessed in Figure 34 where the instance of *RE* is the component *MyRegistry*, which is shared with other activities. Notice that the type of the activity *Bob* is now the activity module in Figure 34, and that the homomorphism now maps *MA* to *BobMortBR*, *RE* to *MyRegistry*, *EM* to *BAM* and *BE* to *BCR*. It is in this sense that the activity is reconfigured as new services are discovered and bound to its requires-interfaces. See [33] for a full formalisation of this process of reconfiguration.

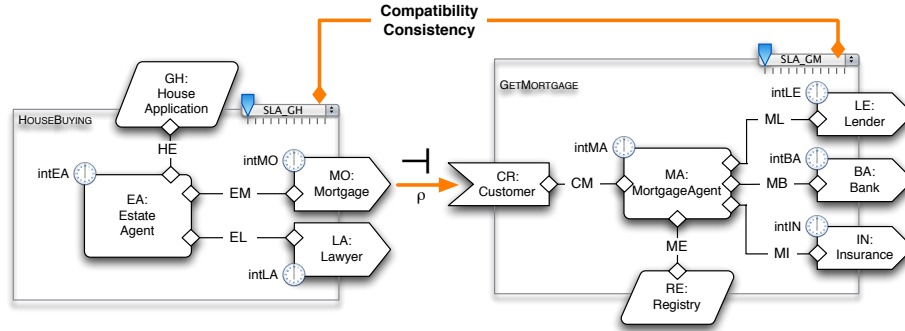


Fig. 32. The elements involved in unification.

7 Checking the Correctness of Service Modules

Service modules are considered to be ‘correct’ when the properties offered in their provides-interface are ensured by the orchestration of their components and the properties specified in their requires-interfaces. Therefore, in order to prove the correctness of GETMORTGAGE, we would need to check that the properties offered through the business protocol *Customer* — e.g., committing to the proposal made by *MA* ensures that a confirmation message will be sent conveying the loan contract — are effectively established by the orchestration performed

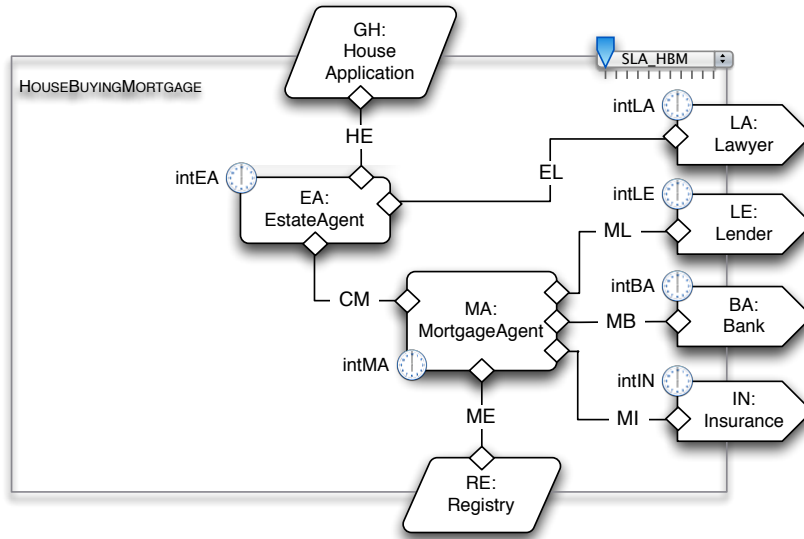


Fig. 33. The result of unification.

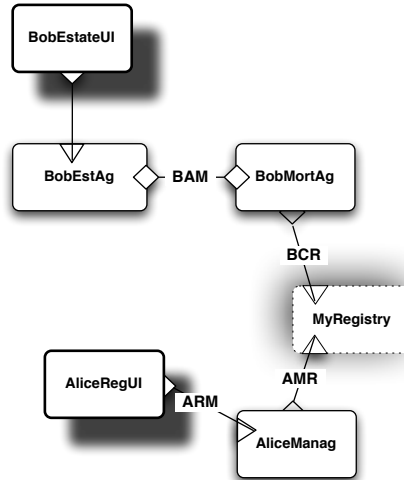


Fig. 34. The result of the binding.

by *MA* on the assumption that the properties required of *LE*, *BA* and *IN* are satisfied.

In this section, we discuss a model-checking approach that we have developed for analysing the properties that can emerge from the orchestration of service behaviour in general, and the correctness of service modules in particular. This approach is based on the model-checker UMC [40] developed at CNR-ISTI. UMC works over UML state machines and UCTL [10], a temporal logic that is interpreted over transition systems in which both states and transitions are labelled, thus making it easier to express properties of stateful interactions as required by SRML.

7.1 The UCTL Semantics of Business Protocols

UCTL is a temporal logic that includes both the branching-time action-based logic ACTL [22] and the branching-time state-based logic CTL [26]. The models of UCTL are doubly labelled transition systems (L^2TS for short) which are transition systems whose states are labelled by atomic propositions and whose transitions are labelled by sets of actions [23]. The syntax of UCTL formulas is defined as follows:

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

where p ranges over *state predicates*, χ over *actions*, ϕ over *state formulae*, and π over *path formulae*. E and A are “exists” and “for all” *path quantifiers* respectively. The next operator X says that in the next state of the path, reached by an action satisfying χ , the formula ϕ holds. The intuitive meaning of the doubly-indexed until operator U on a path is that ϕ' holds at some future state of the path reached by a last action satisfying χ' , while ϕ has to hold from the current state until that state is reached and all the actions executed in the meanwhile along the path either satisfy χ or τ . Finally, the weak-until operator W holds on a path either if the corresponding strong-until operator holds or if for all states of the path the formula ϕ holds and all the actions of the path either satisfy χ or τ . It is straightforward to derive the well-known temporal logical operators EF (“possibly”), AF (“eventually”) and AG (“always”) and the diamond and box modalities $\langle \rangle$ (“possibly”) and \Box (“necessarily”). In particular, $\langle \chi \rangle \phi$ stands for $EX_\chi \phi$, meaning that there is transition that satisfies χ which leads to a state that satisfies ϕ ; and $[\chi]\phi$ stands for $\neg \langle \chi \rangle \neg \phi$, meaning that every transition that satisfies χ leads to a state that satisfies ϕ .

To provide the semantics of business protocols in terms of UCTL formulas, we have to consider the declared set of typed interactions and the set of constraints that correlate the events of those interactions. Recall that the types that are associated with each interaction define not only the set of events the external service can engage in as part of that interaction, but also the conversational protocol that the service follows to engage in those events. We will first address the encoding of the patterns that are used to specify behaviour constraints and then

we will address the encoding of the conversational protocol that is associated with the interaction types.

The semantics of the behavioural patterns used in business protocols (presented in Section 5.1) is defined in terms of UCTL formulas as follows:

initiallyEnabled e	$A \left(true_{\{-e_i\}} W_{\{e?\}} true \right)$
s after a	$AG[a]s$
a enables e	$\left(AG[a] \neg EF < e_i > true \right) \wedge \left(A[true_{\{-e?\}} W_{\{a\}} true \right)$
a ensures e	$\left(AG[a] AF[e!] true \right) \wedge \left(A[true_{\{-e!\}} W_{\{a\}} true \right)$

This encoding is justified by the fact that SRML models correspond to L²TSs in which the actions that label the transitions consist of the several stages of event propagation (publish, deliver, execute or discard), and the state predicates are either pledges (i.e. the properties that ensured by positive replies) or capture the history of events (this is because UCTL does not have past operators).

As already explained, two-way interactions are typed as *s&r* (send and receive) or *r&s* (receive and send) to define that the service being specified engages in the interaction as the requester or as the supplier, respectively. Each of these two roles, requester and supplier, has a set of properties associated with it. The following table presents the UCTL encoding of some of these properties.

s&r — Requester	
The reply-event becomes enabled by the publication of the initiation-event and not before.	$i\Delta! \text{ enables } i\boxtimes?$
r&s — Supplier	
The reply will be published after and only after the initiation-event was executed	$i\Delta? \text{ ensures } i\boxtimes!$
The revoke-event cannot be executed before the execution of the commit-event.	$A[true_{\{-i\ddagger?\}} W_{\{i\checkmark?\}} true]$

7.2 From SRML modules to UML state machines

In order to be able to model-check properties of service behaviour in the context of SRML in general, and the correctness of service modules in particular, we restrict ourselves to those modules in which state machines are used for modelling the internal components, the persistent components, the protocols performed by the wires, and the required behaviour of external services. This is because UMC takes as input a set of communicating state machines with which it associates a L²TS that represents the possible computations of that system. Model-checking is then performed over this L²TS.

As discussed in Section 5, using UML state machines for defining workflows is quite standard. However, the cases of wires and requires-interfaces are not as simple. In the case of wires, we need to ensure that event propagation and related phenomena occur according to the rules of the computational model. In the case of requires-interfaces, we need to discuss how the patterns defined in Subsection 5.1 can be represented with state machines.

Encoding requires interfaces In SRML, requires-interfaces are specified through business protocols with the patterns of temporal logic that we discussed in Subsection 5.1. The proposed encoding associates a state machine with each requires-interface that corresponds to a canonical model of the required behaviour. The strategy of the encoding entails creating a concurrent region for each of the interactions that the external service is required to be involved in — the interaction-regions — and a concurrent region for each of the behaviour constraints — the constraint-regions — except for the constraints defined with the pattern **initiallyEnabled** e : as discussed further ahead, these are modelled by the instantiation of a state attribute.

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 as discussed before. Events of a given interaction are published, executed and discarded exclusively by the interaction-region that models it. The role of the constraint-regions is to flag, through the use of special state attributes, when events become enabled and when events should be published — the evolution of the interaction-regions, and thus the actual execution, discard and publication of events, is guarded by the value of those flags. Constraint-regions cooperate with interaction-regions to guarantee the correlation of events expressed by the behaviour constraints.

We illustrate this methodology by presenting the encoding of the requires-interface *Lender* in Figure 35. *Lender* is involved in the two interactions *requestMortgage* and *requestSignOut*, which are encoded by interaction-regions *A* and *B*, respectively; these two interactions are correlated by two behaviour constraints, the second of which originates the constraint-region *X*. The constraint **initiallyEnabled** *requestMortgage* Δ ? does not originate a region in the state machine; instead it determines that the flag *requestMortgage* Δ _enabled is initially set to *true* and therefore when the event *requestMortgage* Δ is processed it will be executed (and not discarded) by interaction-region *A*. When *requestMortgage* Δ is executed, interaction-region *A* evolves from state *a1* to state *a2* by publishing a positive reply or alternatively from *a1* to the final state by publishing a negative reply. If the commit-event of *requestMortgage* is processed in state *a2*, it will be executed and therefore the *requestMortgage* \checkmark _executed will be set to *true*. It is at this point that the constraint region *X* comes into play — this region reacts to the change of value of *requestMortgage* \checkmark _executed by setting *requestSignOut* Δ _enabled to *true*. After this happens, region *B* will be ready to execute the request-event of *requestSignOut* and therefore this two-way interaction can be initiated.

Following our methodology, each interaction declaration and each behaviour constraint encodes part of the final state machine in a compositional way. Associated with each interaction type, there is a particular statechart structure that encodes it. Each of the patterns of behaviour constraints is also associated with a particular statechart structure. A complete mapping from interactions types and behaviour patterns to their associated statechart structure can be found in [4]. Naturally, the encoding we propose for specifications of requires-interfaces

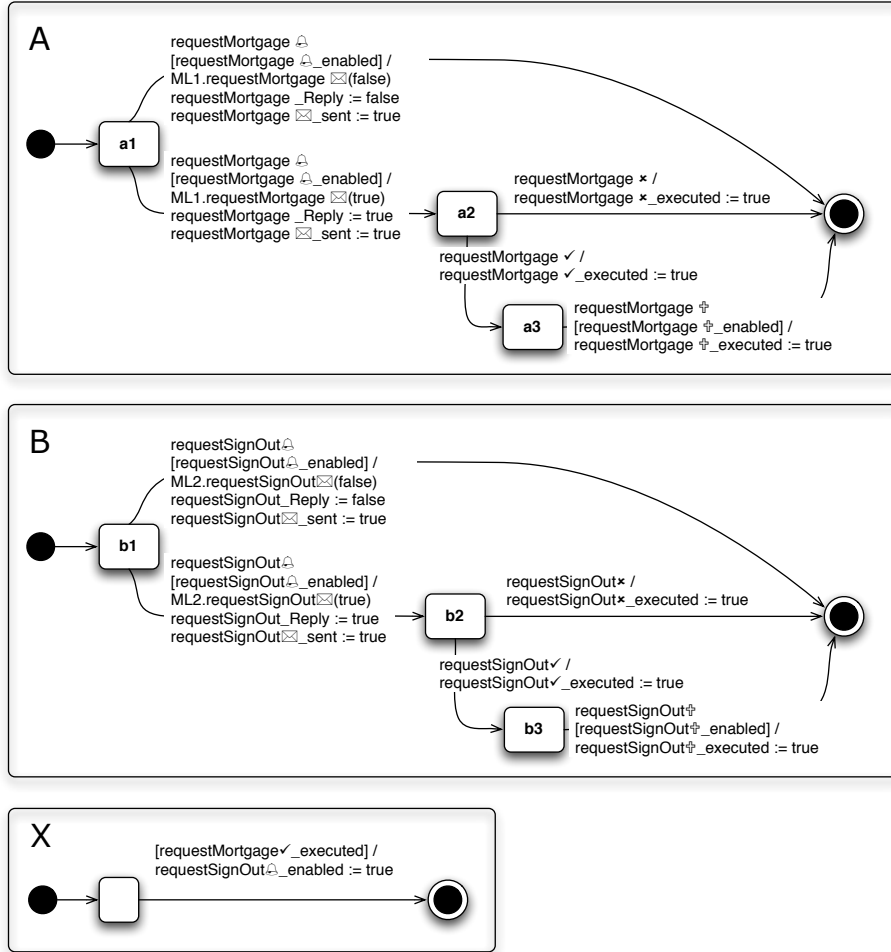


Fig. 35. The UML statechart encoding of the requires-interface *Lender*. *A* and *B* are the interaction-regions and *X* is the constraint-regions.

is defined in such a way that the transition system that is generated for a service module satisfies the UCTL formulas that are associated with each of the requires-interfaces of that module.

Encoding wires In SRML wires are responsible for the coordination of the interactions declared locally for each party of the module. For each wire, there is a connector that defines an interaction protocol with two roles and binds the interactions declared in the roles with those of the parties at the two ends of the wire [5]. With our methodology for encoding wires with UML state machines, every connector defines a state machine for each interaction. This state machine is responsible for transmitting the events of that interaction from the sending party to the receiving co-party. Parties publish events by signaling them in the state machine that corresponds to the appropriate connector; this state machine in turn guarantees that these events are delivered by signaling them in the state machine that is associated with the co-party. The relation between parameter values that is specified by the interaction protocol of the connector is ensured operationally by the state machine that encodes that connector – data can be transformed before being forwarded. The statechart contains a single state and as many loops as the number of events that the connector has to forward.

In GETMORTGAGE, 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 36 shows the state machine that encodes this connector for the single interaction that takes place between *MA* and *LE* — there is only one persistent state in which the machine waits to receive events and forward them with the same parameter values.

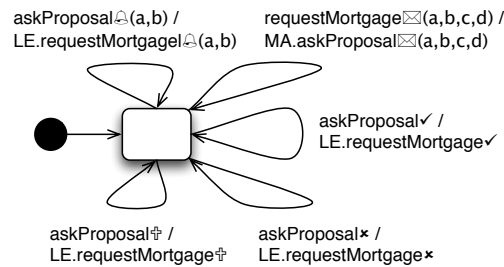


Fig. 36. The UML encoding of the connector that coordinates the single, two-way, interaction between *MA* and *LE* which is named *askProposal* and *requestMortgage* from the point of view of each party respectively.

7.3 Model-checking service modules at work

As mentioned before, our approach to check the correctness of service modules is based on the model-checker UMC [40]. UMC is an on-the-fly model-checker developed for efficient verification of UCTL formulae over a set of communicating UML state machines [43]. A UMC model description consists of a set of UML class definitions and a static set of object instantiations – the actual state machines that form the system under analysis. A UMC model must represent an input-closed system, i.e. the input sources must be modelled as active objects interacting with the rest of the system. Each state machine has a pool that buffers the set of signals that have been received from other machines until they are processed by that machine. According to its class definition, each state machine has at any given time a value for each of its attributes and a set of currently active sub states as specified by the statechart diagram of the class.

In order to illustrate our model-checking approach we will discuss how to model-check the module GETMORTGAGE. First, we have encoded each of its external-required interfaces and each of its connectors using the methodology described in the previous section. Adding the two components that orchestrate the system, we ended up with a set of fourteen communicating UML state machines. Because every input source of a UMC model must also be modelled via an active object, we had to define a machine that initiates the interactions advertised in the provides-interface *Customer*, thus modelling a generic client of the service. Using this system as input to the UMC model-checker, we were able to verify that the doubly labelled transition system that is generated does satisfy the properties associated with the provides-interface *Customer*, shown in Figure 16. As discussed before, these consists of the properties associated with the types of the declared interactions and those that derive from the patterns of behaviour.

8 Analysing Timing Properties of Complex Services

In this Section, we show how SRML can be extended in order to model the delays involved in the business process through which a service is provided and how time-related properties of service-oriented models can be analysed over such models. For instance, we have in mind the ability to certify that the mortgage-brokerage service satisfies properties of the form “*In at least 80% of the cases, a reply to a request for a mortgage proposal will be sent within 7 seconds*”. Properties of this kind are extremely important in a number of application domains and are usually part of the service level agreements (SLAs) that are negotiated between clients and providers. This approach draws from the work reported in [54].

8.1 Timing issues in SRML models

Given two events e_1 and e_2 , we denote by $Delay(e_1, e_2)$ the time that separates their occurrences, e.g. $Delay(getProposal\ominus, getProposal\boxtimes)$ in the example above. Because we wish to adopt the PEPA analysis tools [52,21] (discussed in

another chapter of this book), we assume that such delays follow an exponential distribution of the form $F_{Delay(e_1, e_2)}(t) = 1 - e^{-rt}$. In practical applications, it is rarely the case that it is possible to obtain a complete response-time distribution of all services in the problem under study. It is far more likely that one will only know the average response time. In this setting, it is indeed correct to capture the inherent stochasticity in the system through a *exponential distribution*. The exponential distribution requires only a single parameter, the average response time. Other distributions would require knowledge of higher moments and other parameters which we do not have. We take care not to require too many parameters because finding each one accurately may require careful measurement or estimation. We apply our modelling only in settings where the average response time is a meaningful quantity to use. For example, we do not model systems that have a substantial component requiring a response from a single human participant because the great variance in human response time makes knowledge of the average response time alone insignificant for analysis purposes. This setting connects us to the rich theory of stochastic process including continuous-time Markov chains (CTMC), and a wealth of efficient numerical procedures for their analysis.

In our setting, the rate r is associated with the entity that processes and publishes the events, and used as a modelling primitive in the proposed extension of SRML. Event-based selection of continuations in SRML becomes probabilistic choice in PEPA. We estimate the probability of the relative outcomes and use the resulting probabilities to weight the rates in the PEPA model to ensure the correct distribution across the continuations. In this way all number distributions remain exponential and thus we can achieve probabilistic branching while remaining in the continuous-time Markovian realm.

We report below a number of delays that, according to the computation and coordination model discussed in Section 4.3, can affect service execution. The rates can be negotiated as SLAs with service providers in the constraint systems mentioned in Section 5.2.

Delays in components. Because they may be busy, components store the events they receive in a buffer where they wait until they are processed, at which point they are either executed or discarded. Two kinds of rates are involved in this process:

processingRate. This rate represents the time taken by the component to remove an event from the buffer. Different components may have different processing rates but all events are treated equally by the same component.

executionRate. This represents the time taken by the component to perform the transition triggered by the event, i.e. making changes to the state and publishing events. We assume that discarding an event does not take time. Each transition declared in a business role has its own execution rate, which should be chosen taking into account the specific effects of that transition.

Delays of requires-interfaces. As already mentioned, requires-interfaces represent parties that have to be discovered at run time when the corresponding trigger becomes true. Two kinds of rates are involved in this process:

compositionRate. This rate applies to the run-time discovery, selection and binding processes as performed by the middleware, i.e. (1) the time to connect to a broker, (2) the time for matchmaking, ranking and selection, and (3) the time to bind the selected service. We chose to let different requires-interfaces have different composition rates in order to reflect the fact that different brokers may be involved, depending on the nature of the required external services.

responseRate. These are rates that apply to the responses that the business protocol requires of the external service through statements of the form $e_1 * \text{ensures } e_2!$. More specifically, we consider a rate $\text{responseRate}(e_1, e_2)$ for each such pair of events, which include $\text{responseRate}(a\triangle, a\boxtimes)$ for every interaction a of type **r&s** declared in the business protocol.

Delays in wires. Each wire of a module has an associated transfer rate. As mentioned in Section 2, we are considering only interaction protocols that affect a linear transmission from one party to its co-party, and do not involve complex data transformation.

Delays in synchronous communication and resource contention. The interface of a resource consists of a number of synchronous interactions. We define a synchronisation rate for each such interactions and associate it with the events that resolve synchronisation requests by replying to a query or executing an operation.

In summary, we extend every module M with a time policy P that consists of several collections of rates. Each rate is a term of type $\mathbb{R}^+ \cup \{\top\}$, where \top is the passive rate (i.e., the event with a passive rate occurs only in collaboration with another event, when this second event is ready):

- For every requires-interface $n \in \text{requires}(M)$
 - $\text{compositionRate}(n)$
 - $\text{responseRate}(n)(e_1, e_2)$ for every statement $(e_1 * \text{ensures } e_2!)$
- For every $w \in \text{edges}(M)$, $\text{transferRate}(w)$.
- For every $n \in \text{components}(M)$
 - $\text{processingRate}(n)$
 - $\text{executionRate}(n, P)$ for every transition $P \in \text{trans}(\text{label}_M(n))$
- For every $n \in \text{components}(M) \cup \text{serves}(M) \cup \text{uses}(M)$ and interaction i of type **rpl** and **prf**, $\text{synchronisationRate}(n)(i)$.

The sequence diagram in Figure 37 illustrates how the response time associated with $\text{getProposal}\triangle$ depends on the delays associated with the rates discussed in this section. The value of the rates that apply to components and wires to other components or uses-interfaces are fixed when the module is instantiated, i.e. when the interfaces are bound to components or network connections. The rates that involve requires-interfaces are fixed at run time, subject to SLAs.

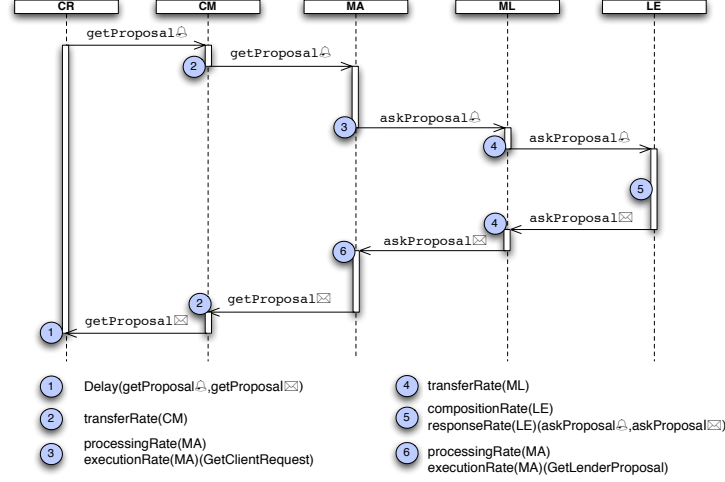


Fig. 37. Cascade of delays in a fragment of *GetMortgage*

8.2 Representing SRML timing issues as stochastic processes

We can now explain how a SRML module can be coded as a stochastic process so that the timing properties that derive from the timing policy of the module can be analysed using PEPA. This encoding involves several steps. First, the structure of the SRML module is decomposed into a PEPA configuration consisting of a number of PEPA terms. Each PEPA term corresponds to either a node or a wire of the original SRML model. In this way we can easily map the results of the quantitative analysis back to the original SRML specification. Second, the behavioural interface of each entity of the SRML model is encoded into a PEPA term, enabling to analyze the delays due to each single component. We use $\langle\langle m \rangle\rangle = t$ to express that the encoding of the SRML element m is the PEPA term t .

Encoding the signature. In SRML the signatures (sets of interactions) associated with specifications of different entities involved in a module are not assumed to be mutually disjoint. This is because we want to promote reuse, which is also why interconnections are established explicitly through wires. Therefore, because in PEPA interconnections are based on shared names, the first step of our encoding consists in renaming all the interactions to guarantee that the interconnections of the SRML model are properly represented by the scopes of action names in PEPA. We do so by defining, for every node n , its encoding signature $esig_M(n)$ obtained by prefixing each interaction name in $sign(label_M(n))$ with n .

The overall encoding $\langle\langle M \rangle\rangle$ of a module M is a cooperation process that includes one sequential component for each node of M , one sequential component

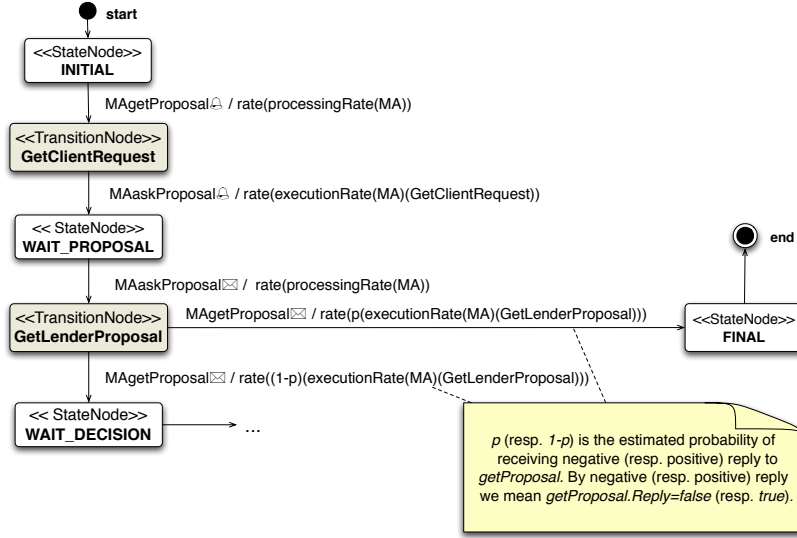


Fig. 38. Statechart for MA with the notation for performance analysis with PEPA

for each edge of M , and one additional sequential component for each requires-interface:

$$\langle\langle M \rangle\rangle = \prod_{n \in \text{nodes}(M)} \langle\langle n \rangle\rangle \boxtimes_{L_1} \prod_{w \in \text{edges}(M)} \langle\langle w \rangle\rangle \boxtimes_{L_2} \prod_{n \in \text{requires}(M)} \langle\langle \text{trigger}_n \rangle\rangle$$

The cooperation set L_1 includes all the interaction events associated with all the interaction names of all the nodes (note that the synchronisation event associated with synchronous interaction types has the same name as the interaction):

$$L_1 = \bigcup_{n \in \text{nodes}(M)} \bigcup_{i \in \text{sign}_M(n)} \{i \ominus i \boxtimes, i \checkmark, i \mathbf{x}, i^\dagger, i\}$$

The cooperation set L_2 includes all the interaction events that act as triggers for requires-interfaces and, for each requires interface n , an event that controls the discovery process associated with n .

$$L_2 = \{m_e : \text{trigger}(n) = (m, e), n \in \text{requires}(M)\} \cup \{\text{discovery}_n : n \in \text{requires}(M)\}$$

Let n be a requires-interface with $\text{trigger}(n) = (m, e)$.

$$\langle\langle \text{trigger}_n \rangle\rangle = P \quad \text{where} \quad P = (m_e, \top).(\text{discovery}_n, \text{compositionRate}(n)). P$$

This term models the delay due to the discovery process that occurs when the trigger becomes true. As shown in Section 8.2, a wire connecting a node to a requires-interface n must wait for the activity discovery_n before enacting any interaction with n .

Encoding components. The PEPA term corresponding to a component-interface n is obtained in two steps: (1) we refine the statechart that defines the business role associated with n , (2) we apply the translation provided by the PEPA toolset [19] to obtain the corresponding PEPA term.

The refinement of the statechart is performed in three substeps. First, the events that occur in the SRML statechart are translated using $esign_M(n)$ as defined in the previous paragraph. That is, given an asynchronous interaction a , $\langle\langle a \ominus \rangle\rangle = n.a \ominus$, $\langle\langle a \boxtimes \rangle\rangle = n.a \boxtimes$, $\langle\langle a \checkmark \rangle\rangle = n.a \checkmark$, $\langle\langle a \mathbf{x} \rangle\rangle = n.a \mathbf{x}$ and $\langle\langle a \ddagger \rangle\rangle = n.a \ddagger$. For every synchronous interaction i , $\langle\langle i \rangle\rangle = n.i$. The second step assigns a probability to the branches of the statechart that are associated with each SRML transition. More precisely, given a transition P with n branches P_{c_i} , we associate a probability p_{c_i} with each branch such that $\sum_{i=1..n} p_{c_i} = 1$. The designer can assign these probabilities taking into account specific knowledge of the application domain, or decide for an equal probability $1/n$, or yet experiment with different values to analyse different possible behaviours. The third step consists in adding the rates. For every SRML transition P of a component n , the incoming arrow is assigned the rate $processingRate(n)$ and each branch P_{c_i} is assigned the rate $p_{c_i} * executionRate(n, P)$. Figure 38 illustrates the statechart diagram for the orchestration of MA , annotated with information on $executionRate$ for each transition.

Wires and interaction protocols. In order to encode a SRML edge $w : n \leftrightarrow m$, we consider first the case when none of the nodes involved is a requires-interface. In this case, all we have to do is to model the transfer of the events from one component to the other. As discussed in Section 2, every wire w defines a set of pairs of interactions $pairs(w)$. We define

$$\langle\langle w \rangle\rangle = \coprod_{(a,b) \in pairs(w)} \langle\langle a, b \rangle\rangle$$

The encoding of the pairs of interactions depends on their types. Consider the case of $\langle \mathbf{s\&r}, \mathbf{r\&s} \rangle$. In this case, the wire forwards the initiation, commit, cancel and revoke events from n to m and the reply back from m to n . We assign the delay $r = transferRate(w)$ to the second leg (delivery to the target):

$$\begin{aligned} \langle\langle a, b \rangle\rangle &= Q \\ Q &= (n.a \ominus, \top).(m.b \ominus, r).Q + (n.a \checkmark, \top).(m.b \checkmark, r).Q + (n.a \mathbf{x}, \top).(m.b \mathbf{x}, r).Q \\ &\quad + (n.a \ddagger, \top).(m.b \ddagger, r).Q + (m.b \boxtimes, \top).(n.a \boxtimes, r).Q \end{aligned}$$

The encoding that applies to the other types of interaction is defined in a similar way. In the case of a one-way asynchronous protocol $\langle \mathbf{snd}, \mathbf{rcv} \rangle$ we have:

$$\langle\langle a, b \rangle\rangle = Q \quad \text{where} \quad Q = (n.a \ominus, \top).(m.b \ominus, transferRate(w)).Q$$

In the case of a synchronous interaction $\langle \mathbf{ask}, \mathbf{rpl} \rangle$ we have:

$$\langle\langle a, b \rangle\rangle = Q \quad \text{where} \quad Q = (n.a, \top).(m.b \ominus, synchronisationRate(m, b)).Q$$

The case of $\langle \text{tll}, \text{prf} \rangle$ is identical. In the case of an edge connecting a requires-interface n , the encoding is:

$$\langle\langle w \rangle\rangle = (\text{discovery}_n, \top). \quad \prod_{\langle a, b \rangle \in \text{pairs}(w)} \langle\langle a, b \rangle\rangle.$$

External-interfaces. The encoding of requires-interfaces n is defined in terms of two processes that cooperate over the set L including all the events in $\text{esign}_M(n)$ and, for each $e \in \text{esign}_M(n)$, the actions enables_e and disables_e . One of the processes (represented by the term S_n) encodes the statements that define the required behaviour of the external party. The other process (represented by the term E_n) controls the enabling and disabling of the interaction events in which the external party can be involved. That is,

$$\langle\langle n \rangle\rangle = S_n \boxtimes_L E_n.$$

Let us consider each process in turn, starting with E_n . We need to control the incoming events, i.e. those received by the external party, all of which have a passive rate. The outgoing events are controlled by the components that receive them through the use of guards as discussed before.

$$E_n = \prod_{\substack{\text{type}(i) \\ = \text{rcv}}} E(n.i \ominus) \prod_{\substack{\text{type}(i) \\ = \text{s\&r}}} E(n.i \boxtimes) \prod_{\substack{\text{type}(i) \\ = \text{r\&s}}} E(n.i \ominus) \mid E(n.i \checkmark) \mid E(n.i \boxtimes) \mid E(n.i \boxplus) \\ E(e) = (\text{enables}_e, \top).E'(e) \quad \text{where} \quad E'(e) = (e, \top).E(e) + (\text{disables}_e, \top).E(e)$$

That is, $E(e)$ synchronises with the enabling of the event, after which it either executes it or disables it again. Consider now the term S_n . We have seen in Section 2.1 that the business protocol associated with a requires-interface n defines a set of statements $\text{statements}_M(n)$. We distinguish three kinds of statements: those that use the connective *initiallyEnabled* – the set of which we denote by A_1 ; those that use *enablesUntil* – the set of which we denote by A_2 ; and those of the form *ensures* – the set of which we denote by A_3 . Each kind of statement is encoded separately, leading to:

$$S_n = \prod_{s_1 \in A_1} \langle\langle s_1 \rangle\rangle \prod_{s_2 \in A_2} \langle\langle s_2 \rangle\rangle \prod_{s_3 \in A_3} \langle\langle s_3 \rangle\rangle \quad \text{where} \quad \langle\langle \text{initiallyEnabled } e_1? \rangle\rangle = \text{enables}_e \langle\langle e_1 \rangle\rangle$$

The enabling action for e_1 has no associated rate (i.e., it is an immediate action, as defined in [9]), because the activity does not involve any of the delays of a SRML module we want to analyze.

$$\begin{aligned} \langle\langle e_1 * \text{enables } e_2? \text{ until } e_3 * \rangle\rangle &= (\langle\langle e_1 \rangle\rangle, \top).P_1 + (\langle\langle e_3 \rangle\rangle, \top).P_2 \\ P_1 &= \text{enables}_e \langle\langle e_2 \rangle\rangle.(\langle\langle e_3 \rangle\rangle, \top).\text{disables}_e \langle\langle e_2 \rangle\rangle. \langle\langle e_1 * \text{enables } e_2? \text{ until } e_3 * \rangle\rangle \\ P_2 &= \text{disables}_e \langle\langle e_2 \rangle\rangle. \langle\langle e_1 * \text{enables } e_2? \text{ until } e_3 * \rangle\rangle \end{aligned}$$

We distinguish between the situation in which e_3 occurs first, disabling e_2 , or e_1 occurs first, enabling e_2 until e_3 occurs. The enabling/disabling are immediate actions.

$$\langle\langle e_1 * \text{ensures } e_2! \rangle\rangle = (\langle\langle e_1 \rangle\rangle, \top).(\langle\langle e_2 \rangle\rangle, \text{responseRate}(n)(e_1, e_2))$$

That is, the execution of e_1 is followed by that of e_2 with a delay whose rate is given by an SLA variable as discussed in Section 3.

Uses-interfaces. As explained in Section 2.1, uses-interfaces provide synchronous interactions with components that offer a certain degree of persistence. For the nodes $n \in \text{serves}(M)$ (notice that synchronous interactions can occur more than once during one module instance):

$$\langle\langle n \rangle\rangle = \sum_{\forall i \in \text{sign}(\text{label}_M(n))} P_{n_i} \quad \text{where} \quad P_{n_i} = (n.i, \text{synchronisationRate}(i)).P_{n_i}$$

8.3 Quantitative analysis of timing properties

Finally, we discuss the quantitative analysis that we are able to perform on a SRML module by using the PEPA Eclipse Plug-in [52] and IPC [21], formal analysis components of the SENSORIA Development Environment³. First, we use the PEPA Eclipse Plug-in tool to generate the statespace of the derived PEPA configuration. We used the static analyser and qualitative analysis capabilities of this tool to determine that the configuration is deadlock free and has no unreachable local states in any component (no “dead code” in the model).

The analysis of a PEPA term encoding a SRML module is inexpensive because the statespace of the model is relatively small, meaning that the number of states of a module grows linearly with respect to the number of nodes. The reason is that the nodes of a SRML module do not execute independently but they wait for one another (i.e., typically not more than one at a time is active).

We performed the passage time analysis of the example illustrated in Figure 37, to investigate the probability of each possible delay between $CRgetProposal\ominus$ and $CRgetProposal\boxtimes$. We conducted a series of experiments on our PEPA model to determine the answers to the following questions:

1. Is the advertised SLA “80% of requests receive a response within 7 seconds” satisfied by the system at present?
2. What is the bottleneck activity in the system at present (i.e. where is it best to invest effort in making one of the activities more efficient?)

The first question is answered by computing the cumulative distribution function (CDF) for the passage from request to response and determining the value at time $t = 10$. The second question is answered by performing a *sensitivity analysis*. That is, we vary each of the rates used in the model (both up from the true value, and down from it) and evaluate the CDF repeatedly over this range of values. The resulting graphs are shown in Figure 39 (the plus denotes the coordinate for 7 seconds and 80%).

Each of the graphs is a CDF which plots the probability of having completed the passage of interest by a given time bound. To determine whether the stated SLA is satisfied we need only inspect the value of this probability at the time bound. For the given values of the rates we find that it is the case that this SLA is not satisfied (Figure 39(a)).

³ Our aim is to discuss the proposed method rather than focusing on the results related to the specific case study, which is used for illustrative purposes.

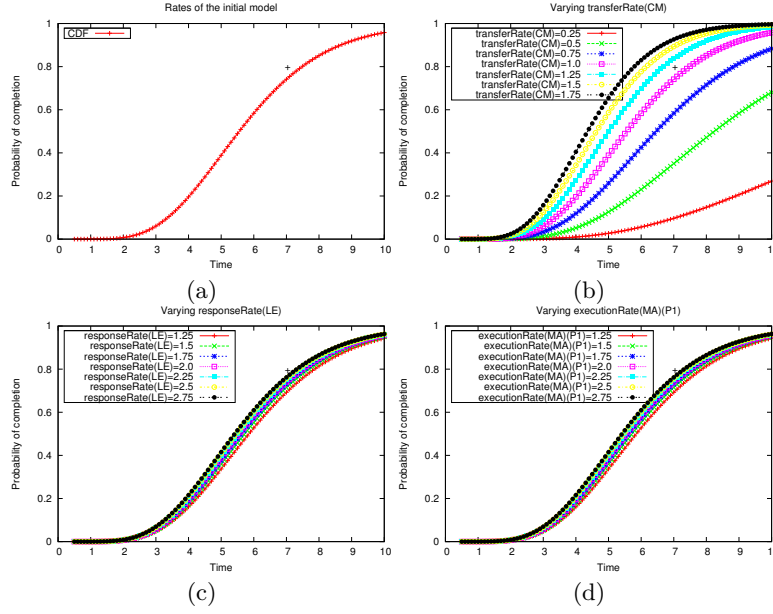


Fig. 39. Sensitivity analysis of response time distributions (from [13])

In performing sensitivity analysis we vary each rate through a fixed number of possible values to see if we can identify an improvement which satisfies the SLA. We have begun by considering seven possible values here. Three of these are above the true value (i.e. the activity is being performed faster) and three are below (i.e. the activity is being performed slower). From the sensitivity analysis we determine (from Figure 39(b)) that variations in rate parameter $transferRate(CM)$ have the greatest impact on the passage of interest. Due to the structure of the model this rate controls the entry into the passage from request to response so delays here have a greater impact further through the passage. In contrast variations in rate parameter $responseRate(LE)$ (seen in Figure 39(c)) and $executionRate(MA)(P1)$ (seen in Figure 39(d)) have the least impact overall. Thus if seeking to improve the performance of the system we should invest in improving $coTransferRate$ before trying to improve $responseTime(LE)$. Figure 39(b) illustrates, for example, how the advertised SLA is satisfied by improving the value of $transferRate(CM)$ to 1.25. It is entirely possible that the sensitivity analysis will identify several ways in which the SLA can be satisfied. In this case the service stakeholders can evaluate these in terms of implementation cost or time and identify the most cost-effective way to improve the service in order to meet the SLA.

9 Related Approaches

One of the main aspects that distinguishes the approach that we proposed from other work on Web Services (e.g. [8]) and SOC in general (e.g. [2]) is that we address not the middleware architectural layers (or low-level design issues in general), but what we call the ‘business level’. For instance, the main concern of the Service Component Architecture (SCA) [2], from which we have borrowed concepts and notations, is to provide an open specification “allowing multiple vendors to implement support for SCA in their development tools and runtime”. This is why SCA offers a middleware-independent layer for service composition and specific support for a variety of component implementation and interface types (e.g. BPEL processes with WSDL interfaces, or Java classes with corresponding interfaces). Our work explores a complementary direction: our research aims for a modelling framework supported by a mathematical semantics in which business activities and services can be defined in a way that is independent of the languages and technologies used for programming and deploying the components that will execute them. The fact that the modelling framework is equipped with a formal semantics makes it possible to support the analysis of services, service compositions and activities, a direction that we are pursuing through the use of model-checking [7].

Another architectural approach to SOC has been designed [53] that follows SCA very closely. However, its purpose is to offer a meta-model that covers service-oriented modelling aspects such as interfaces, wires, processes and data. Therefore, as in SCA, interfaces are syntactic and bindings are established at design time, whereas our interfaces are behavioural and binding occurs at run time. Other approaches to service modelling have considered richer interfaces that encompass business protocols, e.g. [11,28,24,46,47], but not the dynamic aspects — discovery and binding — offered by SRML as illustrated in this paper. Indeed, a characteristic that distinguishes our approach from other formal models of services such as [17] is the fact that we address the dynamic aspects of SOC, namely run-time discovery and binding. Formalisms for modelling (web) services tend not to address these. For example, in BPEL, service compositions are created statically and are governed by a centralised engine. This also holds for approaches that focus on choreography (e.g. [20,46]), where it is possible to calculate which are the partners that can properly interact with a service but the actual discovery and binding processes are not considered. Exceptions can be found among some of the process calculi that have been developed for capturing semantic foundations of SOC (e.g. [30,18,38]). However, such process calculi tend not to address dynamic reconfiguration separately from computation, i.e. the process of discovery and binding is handled as part of the computation performed by a service. As far as we know, SRML is the first service-modelling language to separate these two concerns.

Indeed, in our opinion, what makes SOC different from other paradigms is the fact that it concerns run-time, not design-time complexity. This is also the view exposed in [25] — a very clear account of what distinguishes SOC from CBD (Component Based Development). Whereas in CBD component selection

is either performed at design time or programmed over a fixed universe of components, SOC provides a means of obtaining functionalities by orchestrating interactions among components that are procured at run time according to given (functional) types and service level constraints.

Another area related to the work that we have presented concerns the non-functional aspects of services, namely the policies and constraints for service level agreement that have to be taken into account in the composition of services. Most of the research developed in this area has been devoted to languages for modelling specific kinds of policies (over specific non-functional features) and of selection algorithms, e.g. SCA Policy [2] among several others [41,42,50,49,29]. These languages have been primarily designed to be part of the technology available for implementing and executing services. As such, they are tailored to the technological infrastructure that is currently enabling web services and are not best placed for being used at high-levels of business modelling.

10 Concluding Remarks

In this chapter, we presented an overview of the formal approach for modelling service-oriented application that we developed within SENSORIA towards a methodological and mathematical characterisation of the service-oriented computing paradigm [3]. The approach is built around a prototype language called SRML — the SENSORIA Modelling Reference Language — and offers an engineering environment that includes abstraction mappings from workflow languages (such as BPEL [15]) and policy languages (such as StPowla [14]), model-checking techniques that support qualitative analysis, and stochastic analysis techniques for timing properties. SRML is supported by an Eclipse-based editor (available from www.cs.le.ac.uk/srml) that is part of the SENSORIA Development Environment (SDE). A mathematical semantics is available for all aspects of the approach as partially illustrated in the paper (see [4,6,31,32,33,34] for a more comprehensive account).

This methodology has been tested in a number of other domains, including telco [7], travel [6], automotive [16] and procurement [31] scenarios. Tutorials have been given at CONCUR'08, SEFM'08, SFM'09 and DISCOTEC'09. More extended tutorials were given at the Technical University of Valencia (Spain) and the Summer School on Web Engineering held in 2007 in La Plata, Argentina. SRML is also being taught at the University of Leicester to postgraduate students in Computer Science.

Acknowledgments

We would like to thank our colleagues in the SENSORIA project for many useful discussions on the topics covered in this paper. Stefania Gnesi and Franco Mazzanti (CNR-ISTI) contributed directly to the work presented in Section 7, and Stephen Gilmore (Edinburgh), Monika Solanki (Leicester) and Vishnu Vankayala (Leicester) to Section 8. Artur Boronat and Yi Hong (Leicester) contributed

directly to the development of the SRML Editor. We are also indebted to Colin Gilmore from Box Tree Mortgage Solutions (Leicester) for taking us through the mortgage-brokerage case study.


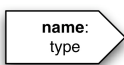
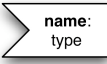

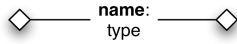

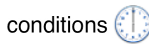
References

1. Global computing initiative. <http://cordis.europa.eu/ist/fet/gc.htm>.
2. The open service oriented architecture collaboration. Whitepapers and specifications available from www.osoa.org (see also oasis-opencsa.org/sca).
3. Sensoria consortium, 2007. White paper available at <http://www.sensoria-ist.eu/files/whitePaper.pdf>.
4. J. Abreu. *Modelling Business Conversations in Service Component Architectures*. PhD thesis, University of Leicester, U.K., 2009.
5. J. Abreu, L. Bocchi, J. L. Fiadeiro, and A. Lopes. Specifying and composing interaction protocols for service-oriented system modelling. In J. Derrick and J. Vain, editors, *Formal Methods for Networked and Distributed Systems, FORTE'07*, volume 4574 of *LNCS*, pages 358–373, 2007.
6. J. Abreu and J. Fiadeiro. A coordination model for service-oriented interactions. In D. Lea and G. Zavattaro, editors, *Coordination Languages and Models*, volume 5052 of *LNCS*, pages 1–16, 2008.
7. J. Abreu, F. Mazzanti, J. Fiadeiro, and S. Gnesi. A model-checking approach for service component architectures. In D. Lee, A. Lopes, and A. Poetzsch-Heffter, editors, *Formal Techniques for Distributed Systems, FMOODS/FORTE '09*, volume 5522 of *LNCS*, pages 212–217, 2009.
8. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services*. Springer, 2004.
9. A. Argent-Katwala, J. Bradley, A. Clark, and S. Gilmore. Location-aware quality of service measurements for service-level agreements. In *Trustworthy Global Computing (TGC'07)*, volume 4912 of *LNCS*, pages 222–239. Springer, 2008.
10. 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.
11. B. Benatallah, F. Casati, and F. Toumani. Web services conversation modeling: A cornerstone for e-business automation. *IEEE Internet Computing*, 8(1):46–54, 2004.
12. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.
13. L. Bocchi, J. Fiadeiro, S. Gilmore, J. Abreu, M. Solanki, and V. Vankayala. A formal approach to modelling time properties of service-oriented systems, 2009. Submitted. (Available from www.cs.le.ac.uk/srml).
14. L. Bocchi, S. Gorton, and S. Reiff-Marganiec. Engineering service-oriented applications: From stpowla processes to srml models. In J. Fiadeiro and P. Inverardi, editors, *Fundamental Aspects of Software Engineering*, volume 4961 of *LNCS*, pages 163–178, 2008.
15. L. Bocchi, Y. Hong, A. Lopes, and J. Fiadeiro. From BPEL to SRML: a formal transformational approach. In M. Dumas and R. Heckel, editors, *Web Services and Formal Methods*, volume 4937 of *LNCS*, pages 92–107, 2007.
16. Laura Bocchi, José Luiz Fiadeiro, and Antónia Lopes. Service-oriented modelling of automotive systems. In *COMPSAC*, pages 1059–1064. IEEE Computer Society, 2008.

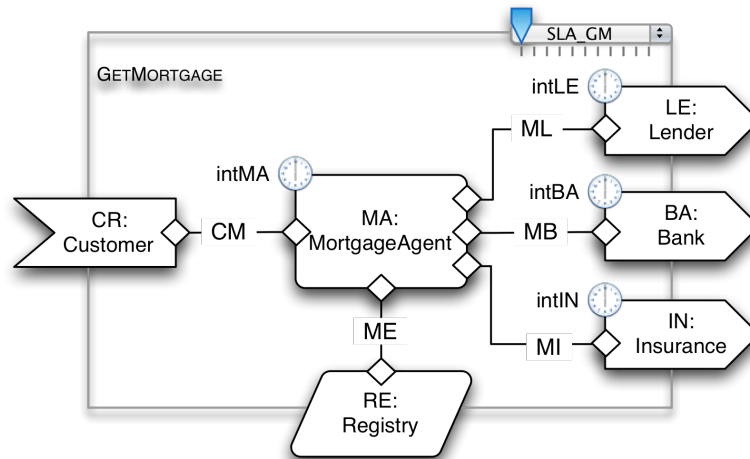
17. M. Broy, I. Kruger, and M. Meisinger. A formal model of services. *ACM TOSEM*, 16(1):1–40, 2007.
18. M. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. In R. De Nicola, editor, *ESOP 2007*, volume 4421 of *LNCS*, pages 18–32, 2007.
19. C. Canevet, S. Gilmore, J. Hillston, M. Prowse, and P. Stevens. Performance modelling with the unified modelling language and stochastic process algebras. In *Computers and Digital Techniques, IEE Proceedings*, volume 150, pages 107–120. IEEE, 2003.
20. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In R. De Nicola, editor, *ESOP 2007*, volume 4421 of *LNCS*, pages 2–17, 2007.
21. A. Clark. The ipclib PEPA Library. In Mor Harchol-Balter, Marta Kwiatkowska, and Miklos Telek, editors, *Proceedings of the 4th International Conference on the Quantitative Evaluation of SysTems (QEST)*, pages 55–56. IEEE, September 2007.
22. R. De Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes*, pages 407–419, 1990.
23. Rocco De Nicola and Frits W. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.
24. R. M. Dijkman and M. Dumas. Service-oriented design: a multi-viewpoint approach. *International Journal of Cooperative Information Systems*, 13(4):337–368, 2004.
25. A. Elfatraty. Dealing with change: components versus services. *Communications of the ACM*, 50(8):35–39, 2007.
26. E.M. Clarke, E.A. Emerson and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
27. G. Coulson et al. A generic component model for building systems software. *ACM TOCS*, 26(1):1–42, 2008.
28. L. Bordeaux et al. When are two web services compatible? In *Technologies for E-Services*, volume 3324 of *LNCS*, pages 15–28, 2005.
29. L. Zeng et al. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.
30. M. Boreale et al. Scc: a service centered calculus. In M. Bravetti, M. Nunez, and G. Zavattaro, editors, *Web Services and Formal Methods*, volume 4184 of *LNCS*, pages 38–57, 2006.
31. J. L. Fiadeiro, A. Lopes, and L. Bocchi. A formal approach to service-oriented architecture. In M. Nunez M. Bravetti and G. Zavattaro, editors, *Web Services and Formal Methods*, volume 4184 of *LNCS*, pages 193–213, 2006.
32. J. L. Fiadeiro, A. Lopes, and L. Bocchi. Algebraic semantics of service component modules. In J. L. Fiadeiro and P. Y. Schobbens, editors, *Algebraic Development Techniques*, volume 4409 of *LNCS*, pages 37–55, 2007.
33. J. L. Fiadeiro, A. Lopes, and L. Bocchi. An abstract semantics of service discovery and binding, 2008. Submitted. (Available from www.cs.le.ac.uk/srml).
34. J. L. Fiadeiro and V. Schmitt. Structured co-spans: an algebra of interaction protocols. In T. Mossakowski, U. Montanari, and M. Haverlaan, editors, *Algebra and Coalgebra in Computer Science*, volume 4624 of *LNCS*, pages 194–209, 2007.
35. I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.
36. Q. Gu and P. Lago. A stakeholder-driven service life-cycle model for soa. In *IW-SOSWE'07*, pages 1–7, 2007.

37. J. Hillston. *A Compositional Approach to Performance Modelling*. 1996.
38. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. volume 4421 of *LNCS*, pages 33–47, 2007.
39. P. Mayer, N. Koch, and A. Schroder. A model-driven approach to service orchestration. In *Proceedings of IEEE International Conference on Services Computing (SCC 2008)*, 2008.
40. 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.
41. N. Mukhi, P. Plebani, I. Silva-Lepe, and T. Mikalsen. Supporting policy-driven behaviours in web services: experiences and issues. In *Proceedings ICSOC’04*, pages 322–328, 2004.
42. A. Mukhija, A. Dingwall-Smith, and D. Rosenblum. Qos-aware service composition in dino. In *ECOWS 2007*, pages 3–12. ACM Press, 2007.
43. Object Management Group. Unified Modeling Language. <http://www.uml.org/>.
44. C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, 2003.
45. J. Rao and X. Su. A survey of automated web service composition methods. In J. Cardoso and A. Sheth, editors, *Semantic Web Services and Web Process Composition*, volume 3387 of *LNCS*, pages 43–54, 2004.
46. W. Reisig. Modeling and analysis techniques for web services and business processes. In *FMOODS 2005*, volume 3535 of *LNCS*, pages 243–258, 2005.
47. W. Reisig. Towards a theory of services. In *UNISCON 2008*, pages 271–281, 2008.
48. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. 1996.
49. K.-J. Lin T. Yu. A broker-based framework for qos-aware web service composition. In *Proc. of the Intl. Conf. on e-Technology, e-Commerce and e-Service*, pages 22–29. IEEE Computer Society, 2005.
50. OASIS WSBPEL TC. Web services business process execution language, 2007. Version 2.0. Technical report, OASIS.
51. M. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An action/state-based model checking approach for the analysis of communication protocols for service-oriented applications. In S. Leue and P. Merino, editors, *Formal Methods for Industrial Critical Systems*, volume 4916 of *LNCS*, pages 133–148.
52. M. Tribastone. The PEPA Plug-in Project. In *Quantitative Evaluation of SysTems*, pages 53–54. IEEE, 2007.
53. W. van der Aalst, M. Beisiegel, K. van Hee, and D. Konig. An soa-based architecture framework. *Journal of Business Process Integration and Management*, 2(2):91–101, 2007.
54. V. Vankayala. Business process modelling using SRML (MSc in Advanced Software Enginnering - Project Dissertation), 2008.

Appendix A —The Iconography

icon	represents	type
	component interface (instantiated when a new session starts; the lifetime is that of the session)	business role (orchestration of interactions)
	requires-interface (bound during service execution after discovery)	business protocol (properties required of external services)
	provides-interface (bound when a new session starts)	business protocol (properties offered by the service)
	uses/serves-interface (bound to a component in the bottom/top layer when a new session starts)	layer protocol (properties assumed of the components in the bottom or top layer)
	wire interface (instantiated together with the second party)	connector (interaction protocol and attachments)
	external configuration policy	constraint system
	internal configuration policy	state conditions

Appendix B —The GetMortgage Service Module



MODULE GETMORTGAGE **is**

DATATYPES

sorts: usrdata, prefdata,
moneyvalue, mortgageproposal,
loandata, loancontract,
insurancedata, accountdata,
setids, bool, nat

PROVIDES



CR: Customer

CR Customer	MA MortgageAgent
r&s getProposal ⌚ idData income preferences ☒ proposal cost	r&s getProposal ⌚ idData income preferences ☒ proposal cost
snd confirmation ⌚ contract	snd confirmation ⌚ contract
SLA VARIABLES CHARGE	SLA VARIABLES CHARGE



REQUIRES

LE: Lender

intLE⌚**trigger:** getproposal⌚?

BA: Bank
 intBA  **trigger:** default
 IN: Insurance
 intIN  **trigger:** default

COMPONENTS

MA: MortgageAgent
 intMA  **init:** s=INITIAL
 intMA  **term:** s=FINAL

USES

RE: Registry

EXTERNAL POLICY

SLA VARIABLES

MA.CHARGE, MA.getProposal[⌚],
 LE.ServiceId, LE.COST, LE.requestMortgage[⌚]

CONSTRAINTS

C₁: {MA.CHARGE, MA.getProposal[⌚]}

$$def(c, t) = \begin{cases} 1 & \text{if } t \leq 10 * c \\ 1 + 2 * c - 0.2 * t & \text{if } 10 * c < t \leq 5 + 10 * c \\ 0 & \text{otherwise} \end{cases}$$

C₂: {LE.ServiceId}

$$def(s) = \begin{cases} 1 & \text{if } s \in MA.lenders \\ 0 & \text{otherwise} \end{cases}$$

C₃: {MA.getProposal[⌚], LE.requestMortgage[⌚]},

$$def(t1, t2) = \begin{cases} 1 & \text{if } t2 > t1 + CM.Delay + ML.Delay \\ 0 & \text{otherwise} \end{cases}$$

C₄: {LE.COST, LE.requestMortgage[⌚]}

$$def(c, t) = \begin{cases} \frac{1}{c} + \frac{t}{100} & \text{if } c < 500 \\ 0 & \text{otherwise} \end{cases}$$

WIRES

MA	C ₄	ME	d ₄	RE
MortgageAgent				Registry
ask getLenders	S ₁	Straight. A(prefdata)R(setids)	R ₁	rpl getLenders
tll regContract	S ₁	Straight. T(loandata, loancontract)	R ₁	prf registerContract

MA MortgageAgent	c_1	MB	d_1	BA Bank
s&r openAccount	S_1	Straight.	R_1	r&s newMortgageAccount
Ⓐ idData	i_1	I(usrdData,	i_1	Ⓐ idData
loanData	i_2	loandata)	i_2	loanData
☒ accountData	o_1	O(accountdata)	o_1	☒ accountData

MA MortgageAgent	c_1	MI	d_1	IN Insurance
s&r getInsurance	S_1	Straight.	R_1	r&s newMortgageInsurance
Ⓐ idData	i_1	I(usrdData,	i_1	Ⓐ idData
loanData	i_2	loandata)	i_2	loanData
☒ insuranceData	o_1	O(insurancedata)	o_1	☒ insuranceData

MA MortgageAgent	c_1	ML	d_1	LE Lender
s&r askProposal	S_1	Straight.	R_1	r&s requestMortgage
Ⓐ idData	i_1	I(usrdData,	i_1	Ⓐ idData
income	i_2	moneyvalue)	i_2	income
☒ proposal	o_1	O(mortgageproposal,	o_1	☒ proposal
loanData	o_2	loandata,	o_2	loanData
accountIncluded	o_3	bool,bool)	o_3	accountIncluded
insuranceRequired	o_4		o_4	insuranceRequired
r&s signOutLoan	S_1	Straight	R_1	s&r requestSignOut
Ⓐ insuranceData	i_1	I(insurancedata,	i_1	Ⓐ insuranceData
accountData	i_2	accountdata)	i_2	accountData
☒ contract	o_1	O(loancontract)	o_1	☒ contract

c_1	CM	d_1	MA MortgageAgent
S_1	Straight.	R_1	r&s getProposal
i_1	I(usrdData,	i_1	Ⓐ idData
i_2	moneyvalue,prefdata)	i_2	income
i_3	O(mortgageproposal,	i_3	preferences
o_1	moneyvalue)	o_1	☒ proposal
o_2		o_2	cost
R_1	Straight	S_1	snd confirmation
i_1	O(loancontract)	i_1	Ⓐ contract

END MODULE

SPECIFICATIONS

LAYER PROTOCOL Registry is

INTERACTIONS

```
rpl getLenders(prefdata):setids
prf registerContract(loandata,loancontract)
```

BEHAVIOUR

BUSINESS ROLE MortgageAgent is

INTERACTIONS

```
r&s getProposal
  ⌚ idData:usrdata,
    income:moneyvalue,
    preferences:prefdata,
  ☒ proposal:mortgageproposal
    cost:moneyvalue
s&r askProposal
  ⌚ idData:usrdata,
    income:moneyvalue,
  ☒ proposal:mortgageproposal
    loanData:loandata,
    accountIncluded:bool,
    insuranceRequired:bool
s&r getInsurance
  ⌚ idData:usrdata,
    loanData:loandata,
  ☒ insuranceData:insurancedata
s&r openAccount
  ⌚ idData:usrdata,
    loanData:loandata,
  ☒ accountData:accountdata
s&r signOutLoan
  ⌚ insuranceData:insurancedata,
    accountData:accountdata,
  ☒ contract:loancontract
snd confirmation
  ⌚ contract:loancontract
ask getLenders(prefdata):setids
tll regContract(loandata,loancontract)
```

SLA VARIABLES

```
CHARGE: [0..100]
```

ORCHESTRATION

```
local    s:[INITIAL, WAIT_PROPOSAL, WAIT_DECISION,
           PROPOSAL_ACCEPTED, SIGNING, FINAL],
         lenders:setids,
         needAccount, needInsurance:bool,
         insuranceData:insurancedata, accountData:accountdata
```

```

transition GetClientRequest
  triggeredBy getProposal⌚
  guardedBy s=INITIAL
  effects s'=WAIT_PROPOSAL
    ∧ lenders'=getLenders(prefdata)
    ∧ ¬empty(lenders') ⊃ s'=WAIT_PROPOSAL
    ∧ empty(lender') ⊃ s'=FINAL
  sends ¬empty(lenders') ⊃ askProposal⌚
    ∧ askProposal.idData=getProposal.idData
    ∧ askProposal.income=getProposal.income
    ∧ empty(lenders') ⊃ getProposal☒
    ∧ getProposal.Reply=false

transition GetLenderProposal
  triggeredBy askProposal☒
  guardedBy s=WAIT_PROPOSAL
  effects needAccount'=askProposal.accountIncluded
    ∧ needInsurance'=askProposal.insuranceRequired
    ∧ askProposal.Reply ⊃ s'=WAIT_DECISION
    ∧ ¬askProposal.Reply ⊃ s'=FINAL
  sends getProposal☒
    ∧ getProposal.Reply=askProposal.Reply
    ∧ getProposal.proposal=askProposal.proposal
    ∧ getProposal.cost=(CHARGE/100+1)*750

transition TimeoutProposal
  triggeredBy now>getProposal.UseBy
  guardedBy s=WAIT_DECISION
  effects s'=FINAL
  sends askProposal*

transition ProposalNotAccepted
  triggeredBy getProposal*
  guardedBy s=WAIT_DECISION
    ∧ now<askProposal.UseBy
  effects s'=FINAL
  sends askProposal*

transition ProposalAccepted
  triggeredBy getProposal✓
  guardedBy s=WAIT_DECISION
    ∧ now<deadline
  effects needAccount ∨ needInsurance ⊃ s'=PROPOSAL_ACCEPTED
    ∧ ¬needAccount ∧ ¬needInsurance ⊃ s'=SIGNING
  sends askProposal✓
    ∧ needAccount ⊃ openAccount⌚
      ∧ openAccount.idData=getProposal.idData
      ∧ openAccount.loanData=getProposal.loanData
    ∧ needInsurance ⊃ getInsurance⌚
      ∧ getInsurance.idData=getProposal.idData
      ∧ getInsurance.loanData=getProposal.loanData
    ∧ ¬needAccount ∧ ¬needInsurance ⊃ signOutLoan⌚
      ∧ signOutLoan.insuranceData=insuranceData
      ∧ signOutLoan.accountData=accountData

transition GetAccount
  triggeredBy openAccount☒
  guardedBy s=PROPOSAL_ACCEPTED
  effects needAccount'=false
    ∧ ¬needInsurance ⊃ s'=SIGNING
    ∧ accountData=openAccount.accountData

```



```

sends ~needInsurance  $\supset$  signOutLoan $\triangleleft$ 
     $\wedge$  signOutLoan.insuranceData=insuranceData
     $\wedge$  signOutLoan.accountData=accountData

transition GetInsurance
  triggeredBy getInsurance $\boxtimes$ 
  guardedBy s=PROPOSAL_ACCEPTED
  effects needInsurance'=false
     $\wedge$  ~needAccount  $\supset$  s'=SIGNING
     $\wedge$  insuranceData=getInsurance.insuranceData
  sends ~needAccount  $\supset$  signOutLoan $\triangleleft$ 
     $\wedge$  signOutLoan.insuranceData=insuranceData
     $\wedge$  signOutLoan.accountData=accountData

transition Conclude
  triggeredBy signOutLoan $\boxtimes$ 
  guardedBy s=SIGNING
  effects s'=FINAL
  sends confirmation $\triangleleft$ 
     $\wedge$  confirmation.contract=signOutLoan.contract
     $\wedge$  regContract(askProposal.loanData,signOutLoan.contract)

```

BUSINESS PROTOCOL Lender is

INTERACTIONS

```

r&s requestMortgage
   $\triangleleft$  idData:usrdata,
    income:moneyvalue,
   $\boxtimes$  proposal:mortgageproposal
    loanData:loandata,
    accountIncluded:bool,
    insuranceRequired:bool
r&s requestSignOut
   $\triangleleft$  insuranceData:insurancedata,
    accountData:accountdata,
   $\boxtimes$  contract:loancontract

```

BEHAVIOUR

```

initiallyEnabled requestMortgage $\triangleleft$ ?
requestMortgage $\checkmark$ ? enables requestSignOut $\triangleleft$ ?

```

BUSINESS PROTOCOL Bank is

INTERACTIONS

```

r&s newMortgageAccount
   $\triangleleft$  idData:usrdata,
    loanData:loandata,
   $\boxtimes$  accountData:accountdata

```

BEHAVIOUR

```

initiallyEnabled newMortgageAccount $\triangleleft$ ?
newMortgageAccount.Reply after newMortgageAccount $\boxtimes$ !

```

BUSINESS PROTOCOL Insurance is

INTERACTIONS

```
r&s newMortgageInsurance
  ⚙ idData:usrdata,
    loanData:loandata,
  ☒ insuranceData:insurancedata
```

BEHAVIOUR

```
initiallyEnabled newMortgageInsurance⚙?
newMortgageInsurance.Reply after newMortgageInsurance☒!
```

BUSINESS PROTOCOL Customer is

INTERACTIONS

```
r&s getProposal
  ⚙ idData:usrdata,
    income:moneyvalue,
    preferences:prefdata,
  ☒ proposal:mortgageproposal
    cost:moneyvalue
snd confirmation
  ⚙ contract:loancontract
```

SLA VARIABLES

```
CHARGE:[0..100]
```

BEHAVIOUR

```
initiallyEnabled getProposal⚙?
getProposal.cost≤750*(CHARGE/100+1) after
  (getProposal☒! ^ getProposal.Reply)
getProposal✓? ensures confirmation⚙!
```

END SPECIFICATIONS