A Formal Approach to Service-Oriented Modelling[†]

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

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

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

Abstract. This paper provides an overview of a formal approach to service-oriented modelling that we have been developing within the SENSORIA project [39]. A modelling language – SRML – and a number of formal techniques that address qualitative and quantitative analysis support this approach, all of which are based on mathematical foundations. Our focus will be on the language primitives that SRML offers for modelling business services and activities, and on the methodological approach that SRML supports.

1 Introduction

Service-oriented computing (SOC) is emerging as the paradigm that will support the new generation of global computers (e.g., the web, the grid), enabling the flexible interconnection of autonomously developed and operated applications [5]. Among the distinguishing aspects of this new paradigm is the fact that the 'construction' of service-oriented applications is focused not so much on the production of code but on service discovery, selection and binding — steps that might occur automatically, at run-time. Because modelling is an essential activity in software construction [18], it is important that developers have available languages that are appropriate for designing applications under this new paradigm.

This paper provides an overview of a formal approach to service-oriented modelling that we have been developing within SENSORIA [39] – an Integrated Project funded under the 'Global Computing' (GC) initiative. A modelling language – SRML – and a number of formal techniques that address qualitative and quantitative analysis support this approach, all of which are based on mathematical foundations. Our focus in this paper is on the language primitives that SRML offers for modelling business services and activities, and on the methodological approach that SRML supports.

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

SRML addresses *Service-Oriented Computing* (SOC) as a new paradigm in which interactions are no longer based on fixed or programmed exchanges of *products* with 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). More precisely, the processes of discovery and selection of services as required by an application are not coded (at design time) but performed by the middleware according to functional and non-functional requirements (SLAs). The process of binding the client application and the selected service is not performed by skilled software developers, but also at run time, by the middleware. Because the set of available services changes as providers update their portfolios, and service-level agreements may be context-dependent, different instances of the same application may bind to different services and operate according to different SLAs resulting from different negotiations.

SOC brings to the front many aspects that have already been discussed about component-based development (CBD), for instance in [24]. 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 decide to procure an external courier service 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. Another important difference can be found in the nature of the interactions, which in SOC tend to be 'conversational' in the sense that they follow a message-exchange protocol that reflects a business-oriented mode of composition [7].

One of our key concerns in defining SRML was precisely the need to distinguish between these two different modes of composition (as further discussed in Section 4). In what concerns SOC, we decided to follow the basic principles and structures put forward by the Service Component Architecture (SCA) [46]. However, SCA addresses low-level design in the sense that it provides an assembly model and binding mechanisms for service components and clients programmed in specific languages, e.g. Java, C++, BPEL, or PHP. Instead, what we find in SRML are primitives that address high-level design and support a shift of emphasis from programming to (business) modelling, from component interoperability to business integration. This means that we will be discussing SOC at a level of abstraction that is different from most other work on Web services (e.g. [7, 37]) and Grid computing (e.g. [29]), including the languages and standards that have been put forward by organisations such as OASIS (www.oasis-open.org) and W3C (www.w3.org). SRML is 'technology agnostic' in the sense that it does not commit to any specific language or platform for programming and composing services.

Several aspects of (earlier versions of) SRML and its associated methodology have already been presented at a number of conferences and workshops (e.g. [2,10,25]). This paper integrates all these aspects and presents a coherent view of the language in its final version. Although our approach is formal, details of its mathematical semantics have been left out; they can be found instead in [1,27] (see also the shorter accounts of fragments of this semantics that have been published in [3,26,28]). Therefore, the paper is mostly mathematics-free with the exception of Section 3.2 and parts of Section 5.

The paper proceeds as follows. In Section 2, we provide an 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 3, we put forward the coordination model on which SRML is based. In Section 4, we present the modelling primitives of SRML. In Section 5, we discuss our model of configuration management. Finally, in Section 6, we discuss related approaches. As a running example, we use a mortgage brokerage service that is part of a larger financial case study developed within SENSORIA.

2 From Use-Case Diagrams to Service and Activity 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 specificities of service-oriented software engineering. 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 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.

2.1 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. 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 – that can be bound to other software components either statically (in a component-based way) or dynamically (in a service-oriented way):

- *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, which are triggered 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).
- *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 4.1.3 we call 'business protocols' – that are organised in a hierarchical ontology to facilitate discovery.

The distinction between these two kinds of applications reflects the existence of different stakeholders in service-oriented development (see [31] for a wider discussion) and has two important methodological implications. On the one hand, 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, the services and activities to be developed by an organisation constitute logical units.

In our example, *UPDATEREGISTRY* is 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 web interface). On the other hand, *GETMORTGAGE* is meant to be placed in a service repository for being discovered and bound to activities 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 for use-case diagrams to reflect the methodological implications of our approach, we proposed in [10] a number of extensions to the standard notation. Figure 1 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 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.

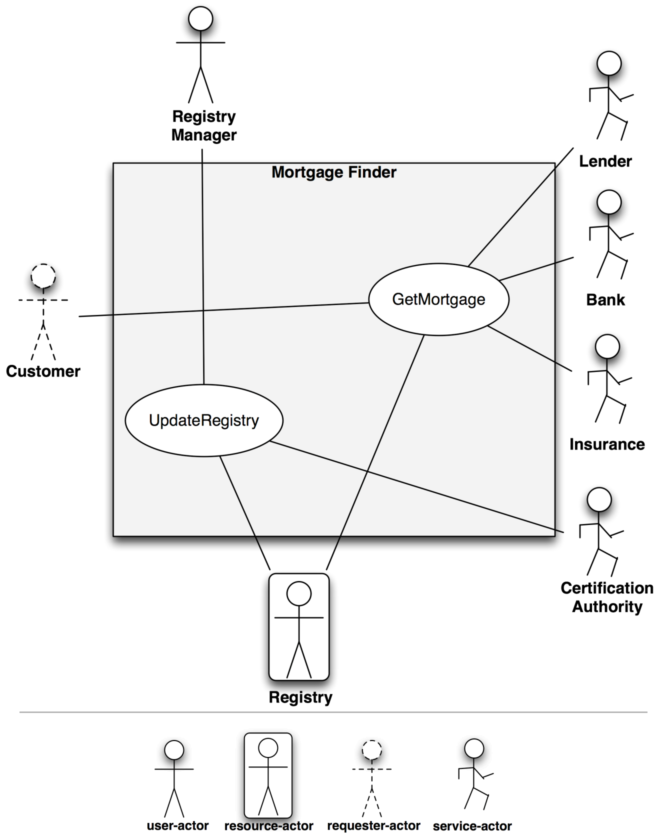


Figure 1: Service-oriented use-case diagram for *Mortgage Finder*

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 1. 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, these descriptions are given by business protocols (discussed in Section 4.1.3) and organised in a shared ontology, which facilitates and makes the discovery of business partners more 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. Such organisational aspects are explored in [12] from the point of view of virtual organisation breeding environments in the sense of [19].

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 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/invocation. 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).

2.2 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 2 presents the structure of the modules derived from the use-case diagram in Figure 1.

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 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 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 SLA), which consists of the variables and constraints that determine the quality profile of the activity to which the discovered services need to adhere.

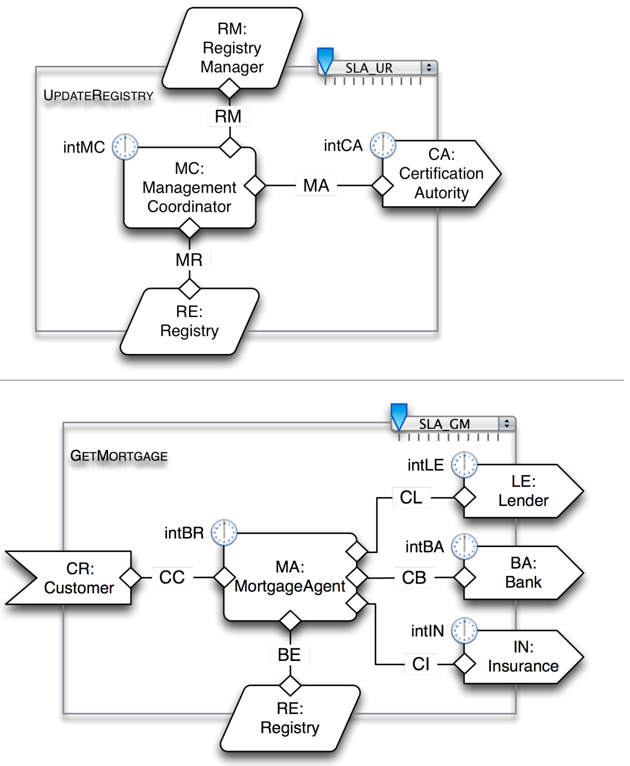


Figure 2: The SRML modules for the activity *UPDATEREGISTRY* and the service *GETMORTGAGE*

The language primitives that are used in SRML for defining all these interfaces as well as the configuration policies are detailed in Section 4. A summary of the graphical notation can be found in Appendix A at the end of the paper.

3 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 [1].

3.1 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 to reflect the typical business conversations that arise in SOC [7]. 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 can be in both directions, i.e. they can be conversational. Interactions are described from the point of view of the party in which they are declared, i.e. ‘receive’ means invocations received by the party and sent by the co-party, and ‘send’ means invocations made by the party. Interactions can be synchronous, implying that the party waits for the co-party to reply or complete, or asynchronous, in which case the party does not block. Typically, synchronous (blocking) interactions occur with persistent components, reflecting interconnections based on the exchange of *products* (clientship as in OO). Interactions among the (internal) components responsible for the orchestration are non-blocking so that they can engage in multiple, concurrent conversations with different parties.

Interactions of type r&s and s&r are conversational, i.e. they involve a number of events exchanged between the two parties:

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

The meaning of the these events should be self-explanatory: 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 conversational 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 occurs, the cancel-event occurs or the validity period associated with the offer expires, whichever happens first. We denote the pledge associated with *interaction* by *interaction*⌛ and its validity period by *interaction*💣. See Figure 3 for some of the possible scenarios (explained further below) that can arise when the co-party replies positively and offers a deal.

All interactions can have parameters for transmitting data when they are initiated – declared as 🔔. Conversational interactions can also have parameters for carrying a reply – declared as ✉ – or for carrying data if there is a commit, a cancel or a revoke – declared as ✓, ✗ and ✞ respectively. In particular, every reply-event *interaction*✉ 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, or negative, in which case the interaction terminates. That is, *interaction.Reply* is *false* if, for some reason related with the business logic, the request *interaction*🔔 can not be fulfilled.
- *UseBy* is a parameter that, in the case of a positive reply, indicates the expiry date of the pledge (i.e. the deal being offered). The value of this parameter (including the value $+\infty$) is obtained by adding the value of the configuration variable (non-functional attribute) *interaction*💣 to the instant at which *interaction*✉ is sent. As discussed in Section 4.2.2, 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 3 illustrate the protocol 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*✓ (this can however be constrained by the business logic, for instance, by defining a deadline for the compensation to be available). In the middle, there is a cancellation; in this situation, a revoke is not available. On the right, the expiry time occurs without a commit or cancel having occurred; this implies that no further events for that interaction will occur. In Section 4.1, we give examples of the intended usage of these primitives.

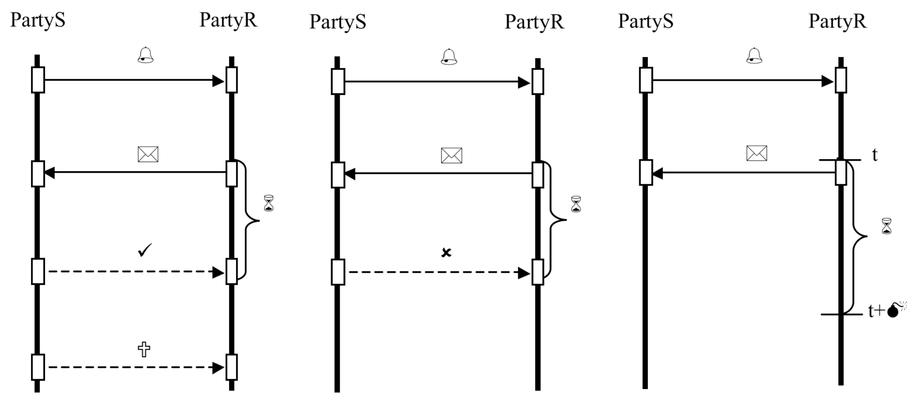


Figure 3: The protocol of conversational interactions when the reply is positive

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¿*) for its execution (resp. being discarded) by the party that receives it. The occurrences of *event!* and *event? (or event¿)* 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 4). In this paper, we do not explore in any depth the use of such delays. See instead [9] for a formal model over which timing aspects of service provision can be analysed through a tool like PEPA [32].

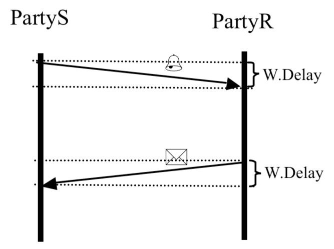


Figure 4: The intuitive semantics of delays.

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 5 concerns the workflow that is initiated by the initial request received by *GETMORTGAGE* from the customer *CR*.

3.2 A formal model

The coordination model of SRML, summarised below, is defined in terms of states and state transitions (see [1] for a detailed definition). 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 <u>state</u> 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;
- 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 (¡), executed (?) or discarded (¿);
- The values of all event parameters and configuration attributes.

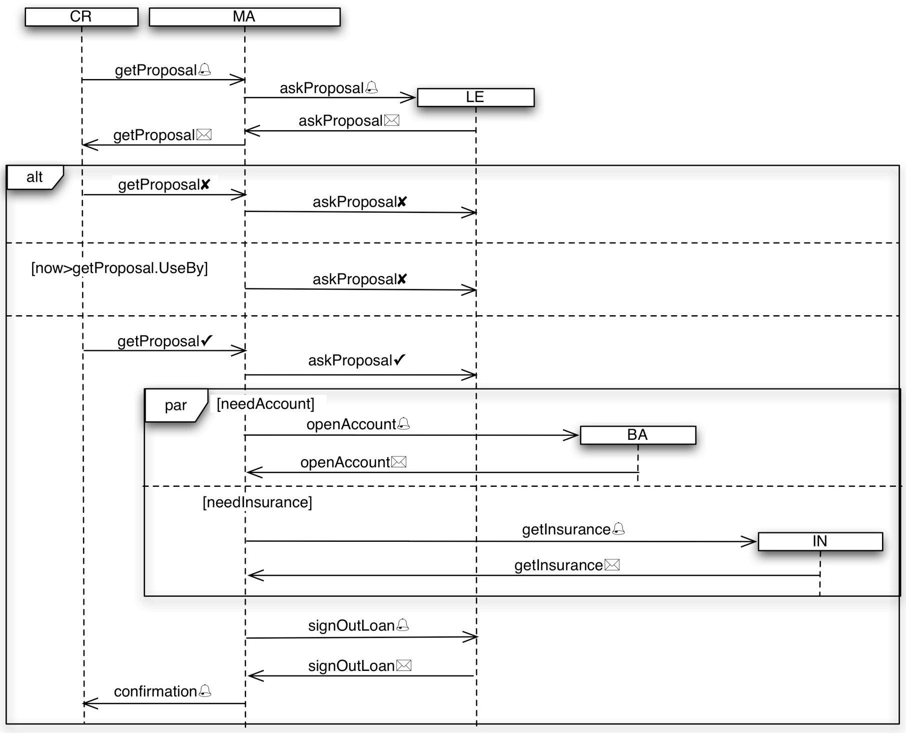


Figure 5: Identifying interactions within GETMORTGAGE.

In this model, state transitions are characterised by what we call a computation step.

A <u>computation step</u> consists 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* consisting of the events that are published during that step together with a function that assigns a value to the parameters of each such event;

such that:

- The set INV^{TRG} of the events in the target state that have been invoked are those in *DLV* (i.e. 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 are those in *PUB* (i.e. published during the step) together with those of 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 [1] for a model that considers unreliable wires.

At each step, components may choose to process one of the events waiting to be processed; this is captured by the function *PRC*. The fact that 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 6 is a graphical 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.

4 The Modelling Primitives of SRML

4.1 Behaviour specification languages

The entities involved in service and activity modules – component interfaces, requires-interfaces, provides-interfaces, uses-interfaces, serves-interfaces and interaction protocols – 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. The full specification is available in Appendix B.

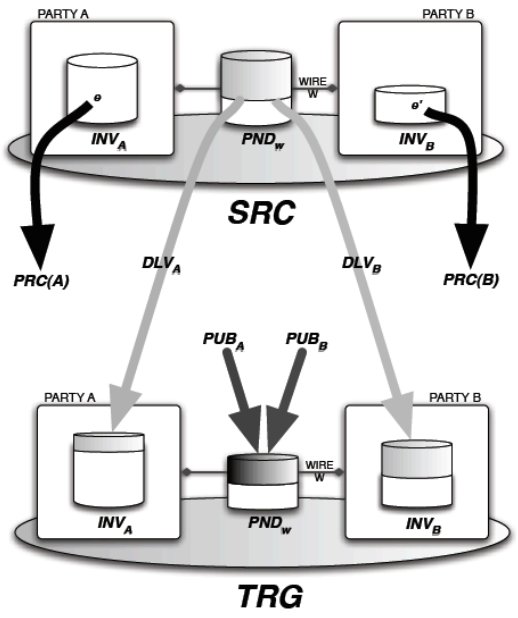


Figure 6: The event flow from the point of view of a wire W between parties A and B.

4.1.1 Signatures

All the languages that we use have in common the declaration of the interactions (in the sense of Section 3.1) 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, which is precisely 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:

```
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)
```

Interactions are classified according to the types defined in Section 3.1. For instance, *getProposal* is declared to be of type 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 🔔. Parameters that are used by the mortgage agent for sending the reply are declared under the symbol ✉ – 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 3.1). 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 seven additional interactions, all of which are self-initiated. While *askProposal*, *getInsurance*, *openAccount*, *signOutLoan* and *confirmation* are asynchronous (i.e. of type s&r or 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) whereas, 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.

4.1.2 Business roles

In SRML, interfaces to 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 actions 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 follows:

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

We use *s* 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. For instance:

```
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
```

- 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 occur – in *GetClientRequest*, the state *INITAL*. 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 on the local state. Given a state variable *var*, we use *var'* to denote the value that it takes after the transition. In the case above, 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.

Another example of a transition is *GetLenderProposal*:

```
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
```

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 4.2.2. 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 4.2.2 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.

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

Notice that, through business roles, SRML offers a very flexible way to model 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.

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

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 transitions are specified as sentences using a logical notation.

More traditional (control-oriented) notations can be used instead for defining orchestrations. In Figure 7 we show how part of the orchestration of *MortgageAgent* can be defined using a UML statechart. Because statecharts focus only on control flow, we would need to provide a separate specification for the data flow. In [14], we have also shown how fragments of BPEL can be encoded in our language.

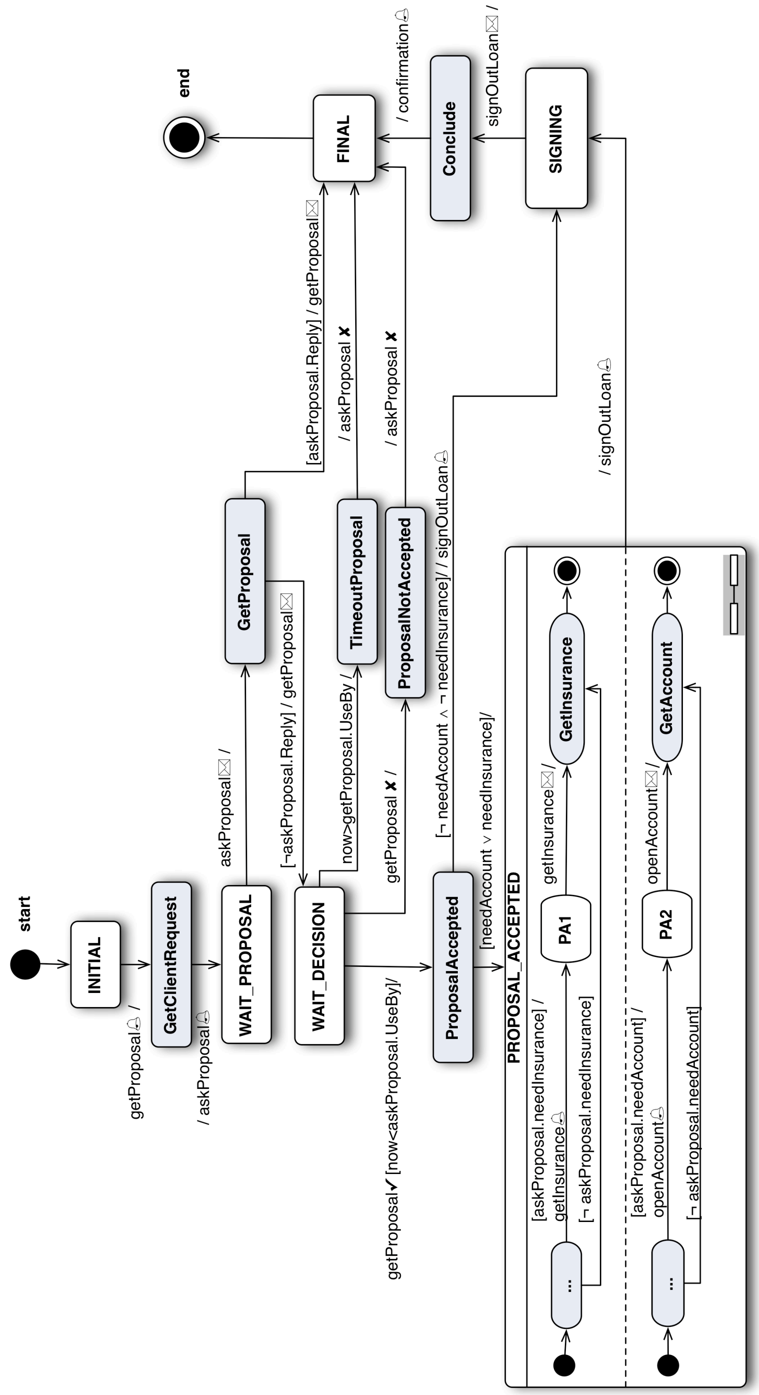


Figure 7: Using UML statecharts for defining orchestrations in business roles

4.1.3 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 following behaviour from a lender:

```
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🔔?
    requestSignOut✉.Reply after requestSignOut✉?
```

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 reply to *requestSignOut* is always positive.

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. Their semantics have been defined in terms of formulas of the temporal logic UCTL [6]. Intuitively, they correspond to traces of the form depicted in Figure 8:

- ***initiallyEnabled*** *e*: The event *e* is enabled (cannot be discarded) in the initial state and remains so until it is executed.
- *a* ***after*** *e*: *a* holds forever after event *e* is executed.
- *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).
- *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.

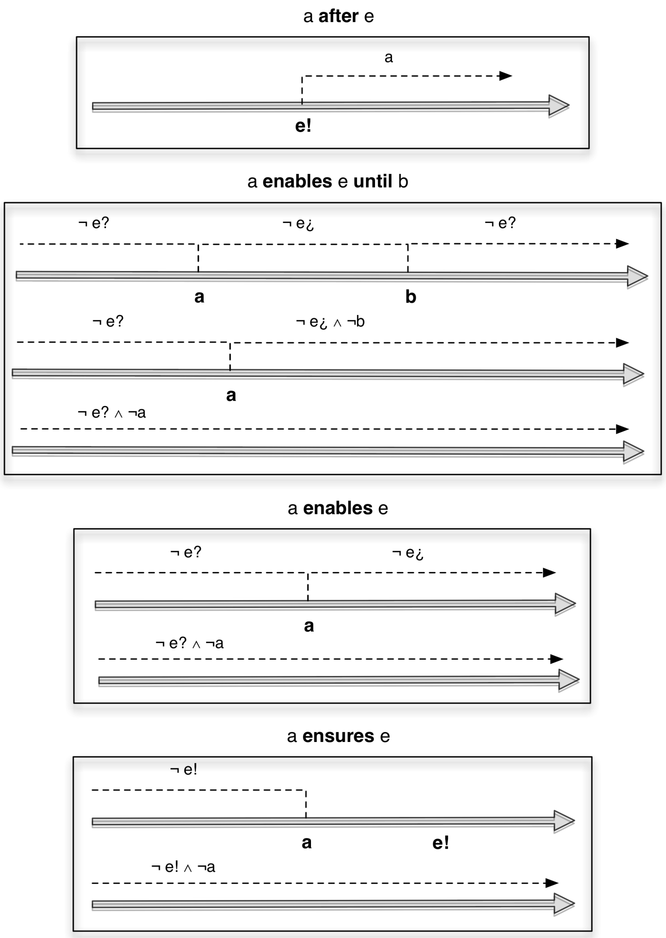


Figure 8: The traces that correspond to the patterns

Business protocols are also used for modelling the behaviour that requesters can expect from a service. This subsumes what, in [5], 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 following business protocol:

```
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🔔!
```

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*.

4.1.4 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 2.1 – 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 following specification of the layer protocol fulfilled by a registry. 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*.

```
LAYER PROTOCOL Registry is
    INTERACTIONS
        rpl getLenders(prefdata):setids
        prf registerContract(loandata,loancontract)
```

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

4.1.5 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. This may include routing events, superposing protocols for secure communication, or transforming sent data to the format expected by the receiver, inter alia.

As an example, consider the following protocol used in the wire that connects *MortgageAgent* and *Insurance*:

```
INTERACTION PROTOCOL Straight.I(d1,d2)O(d3) is
    ROLE A
        s&r S1
            🔔 i1:d1, i2:d2
            ✉ o1:d3
    ROLE B
        r&s R1
            🔔 i1:d1, i2:d2
            ✉ o1:d3
    COORDINATION
        S1 ≡ R1
        S1.i1=R1.i1
        S1.i2=R1.i2
        S1.o1=R1.o1
```

This is a 'straight' protocol that connects directly two entities over two conversational interactions that have two 🔔-parameters and one ✉-parameter. The property $S_1 \equiv R_1$ establishes that the events associated with each interaction are the same, for example 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 in Section 4.1.6.

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.

```
INTERACTION PROTOCOL Straight.A(d1,d2)R(d3) is
   ROLE A
      ask S1(d1,d2):d3
   ROLE B
      rpl R1(d1,d2):d3
   COORDINATION
      S1(d1,d2)=R1(d1,d2)
```

```
INTERACTION PROTOCOL Straight.T(d1,d2) is
   ROLE A
      tll S1(d1,d2)
   ROLE B
      prf R1(d1,d2)
   COORDINATION
      S1(d1,d2)≡R1(d1,d2)
```

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.

4.1.6 Connectors

After having chosen the protocols that coordinate the interactions between two parties, we use them as the 'glue' (in the sense of [40]) 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 $<\mu_A,P,\mu_B>$ 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.

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

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

In the specification of a wire, 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 interactions 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 *CL* (with *MA*). Its specification is as follows:

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(usrdata, 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

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 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 *CC* that interconnects *Customer* and *MortgageAgent* in GETMORTGAGE module is specified as follows:

c_1	**CC**	d_1	**MA** MortgageAgent
S_1	Straight.	R_1	**r&s** getProposal
i_1	I(usrdata,	i_1	🔔 idData
i_2	moneyvalue,prefdata)	i_2	income
i_3	O(mortageproposal,	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

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*'s 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*.

4.2 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 the 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 initialisation 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.

4.2.1 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*🔔 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.

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

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*:

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

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.

4.2.2 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 pledge is valid after *interaction*✉ is issued.

- *wire.Delay*, for every wire; it defines the maximum delivery delay for events transmitted through 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, in *GETMORTGAGE* we have:

```
SLA VARIABLES
  MA.CHARGE, MA.getProposal💣,
  LE.ServiceId, LE.COST, LE.requestMortgage💣
```

The approach that we adopt in SRML for SLA negotiation is based on the constraint satisfaction and optimization framework presented in [8], 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 $<[0..1],max,min,0,1>$ of soft fuzzy constraints [8]. In this c-semiring, the preference level is between 0 (worst) and 1 (best).

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

C_1: *{MA.*CHARGE*,MA.getProposal*💣*}*,

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

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

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*💣*,LE.requestMortgage*💣*}*,

$$def(t1,t2)= \begin{cases} 1 \ \ if \ \ t2 > t1 + CC.Delay + CL.Delay \\ 0 \ \ otherwise \end{cases}$$

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 *CC.Delay* and *CL.Delay* are not declared as SLA variables and, hence, they are used like constants).

C_4: *{LE.*COST*,LE.requestMortgage*💣*}, def(c,t)=* $\begin{cases} \frac{1}{c} + \frac{t}{100} \ \ if \ \ c < 500 \\ 0 \ \ otherwise \end{cases}$

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 the service discovery/binding. Details on negotiation of constraints and SLAs are further discussed in Section 5.3.

4.3 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 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 of 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 follows:

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

Formally:

A service module *M* consist of:

- A graph *graph(M)*.
- A distinguished subset of nodes *requires(M)⊆nodes(M)*.
- A distinguished subset of nodes *uses(M)⊆nodes(M)*.
- A node *provides(M)∈ nodes(M)* distinct from *requires(M)* and *uses(M)*.
- A labelling function $label_M$ such that
  - o $label_M(n)$∈***BROL*** if *n∈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)*.
  - o $label_M(n)$∈***BUSP*** if *n∈requires(M)*
  - o $label_M(provides(M))$∈***BUSP***
  - o $label_M(n)$∈***LAYP*** if *n∈uses(M)*
  - o $label_M(e:n\leftrightarrow m)$∈***CNCT***.
- An internal configuration policy.
- An external configuration policy.

5 The Configuration-Management Model

In this section, we provide an overview of the model that supports the dynamic aspects of SOC as captured in SRML. Full details of the mathematical semantics can be found in [27].

5.1 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. [40]), 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 9 for an example, over which we will later recognise three business activities (instances).

Summarising, a state configuration $\mathcal{F}$ is defined to consist of:

- 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*($\mathcal{F}$)$\subseteq$***COMP*** (i.e. nodes are components) and *edges*($\mathcal{F}$)$\subseteq$***WIRE*** (i.e. edges are wires).
- A (configuration) state $\mathcal{S}$ as defined in 3.2.

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 10). This can be noticed in Figure 9 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 2 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), 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.

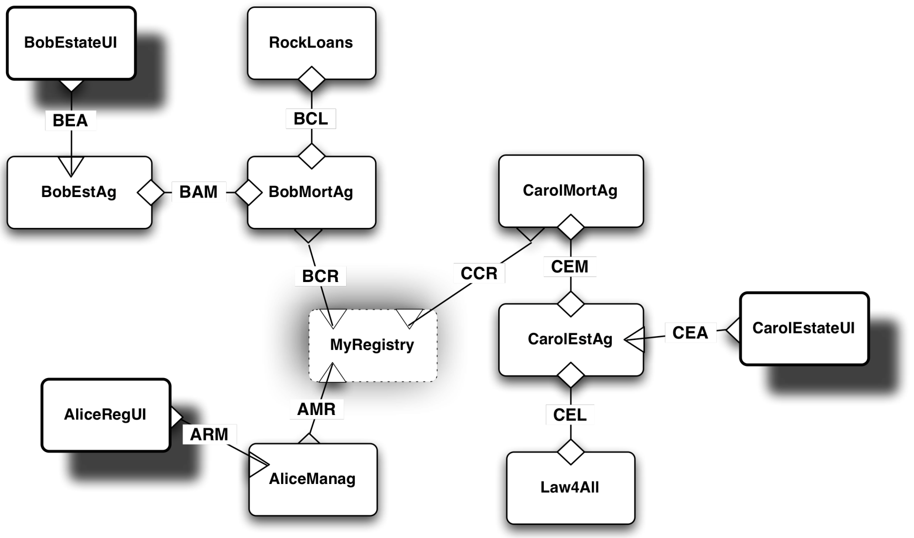


Figure 9: The graph of a state configuration with 11 components and 10 wires

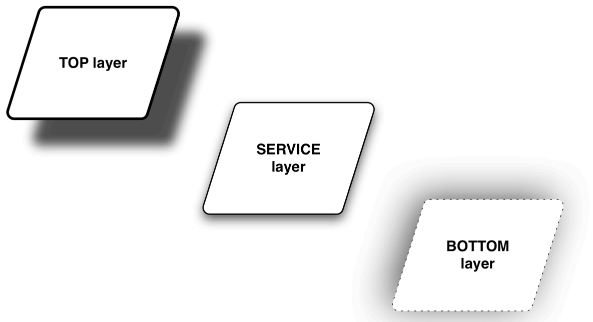


Figure 10: A 3-layered architecture for configurations

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 mortgage 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 2 – 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.

5.2 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 components 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.

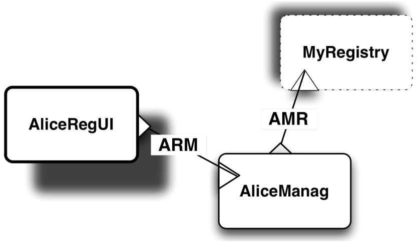


Figure 11: The sub-configurations corresponding to an instance of *UPDATEREGISTRY*

For instance, we should be able to recognise an activity in Figure 9 whose sub-configuration is as depicted in Figure 11. Intuitively, it corresponds to an instance of *UPDATEREGISTRY*. In order to formalise this notion of typed subconfiguration, we start by providing a formal definition of activity modules. We denote by ***BROL*** the set of business roles (see 4.1.2), by ***BUSP*** the set of business protocols (see 4.1.3), by ***LAYP*** the set of layer protocols (see 4.1.4), and by ***CNCT*** the set of connectors (see 4.1.6).

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

- $label_M(n)\in$***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)*.
- $label_M(n)\in$***BUSP*** if *n*$\in$*requires(M)*
- $label_M(n)\in$***LAYP*** if *n*$\in$*serves(M)*$\cup$*uses(M)*
- $label_M(e{:}n\leftrightarrow m)\in$***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.

We can now formalise the typing of state configurations with activity modules that we discussed around Figure 11, 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 $\mathscr{A}$ 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 $\mathscr{F}$.
- A partial mapping $\mathscr{B}$ that assigns an activity module $\mathscr{B}(a)$ to each activity $a\in\mathscr{A}$ – the workflow being executed by *a* in *SF*. We say that the activities in the domain of this mapping are those that are active in that state.
- A mapping $\mathscr{G}$ that assigns an homomorphism $\mathscr{G}(a)$ of graphs $body(\mathscr{B}(a))\rightarrow\mathscr{F}$ to every activity $a\in\mathscr{A}$ that is active in $\mathscr{F}$. We denote by $\mathscr{F}(a)$ the image of $\mathscr{G}(a)$ – the sub-configuration of $\mathscr{F}$ 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. 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. For instance, in the example discussed above, we have an activity – that we call *Alice* – for which $\mathscr{B}(Alice)$ is *UPDATEREGISTRY* (as in Figure 2), $\mathscr{F}(Alice)$ is the sub-configuration in Figure 11, and $\mathscr{G}$ 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: requires-interfaces are used 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 bottom layer of the state configuration.

5.3 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 12.

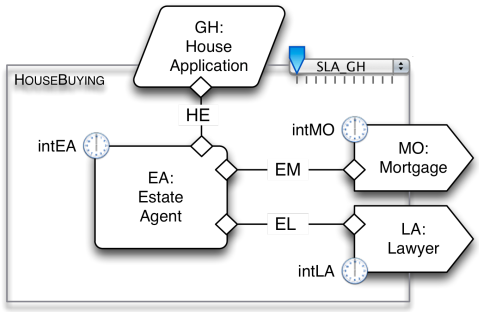


Figure 12: The *HOUSEBUYING* activity module

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*.

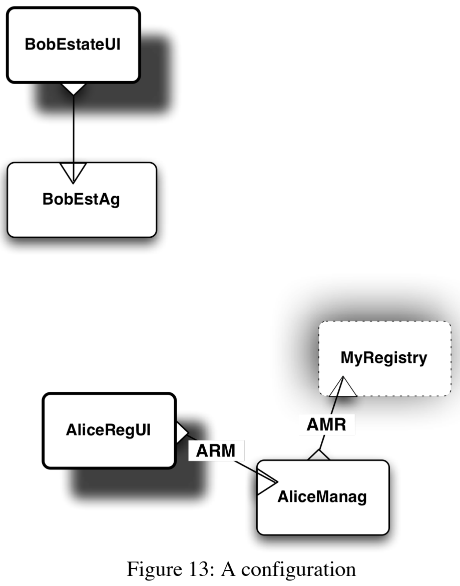


Figure 13: A configuration

Consider the configuration depicted in Figure 13, and the business configuration that consists of *Alice* (as defined in Section 5.2) and of the activity *Bob* typed *HOUSE-BUYING* 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 14).

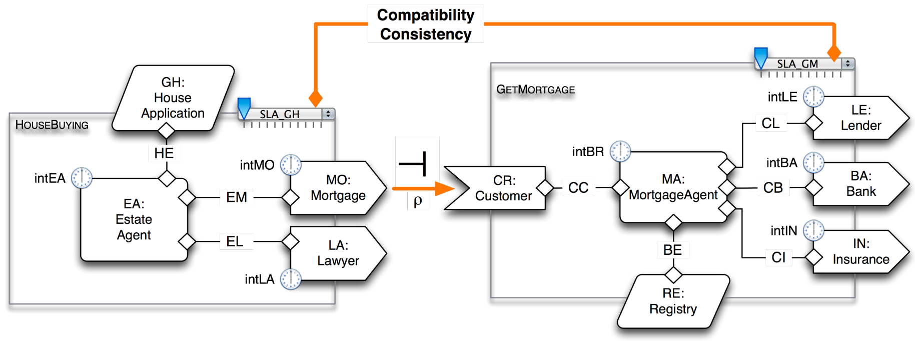


Figure 14: The elements involved in unification

In our setting, this process involves three steps, outlined as follows:

- 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 [27]. If *GETMORTGAGE* is selected then it is unified with *HOUSEBUYING*, giving rise to another activity module. As depicted in Figure 15, 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 16 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 16, 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. A full formalisation of this process of reconfiguration is also presented in [27].

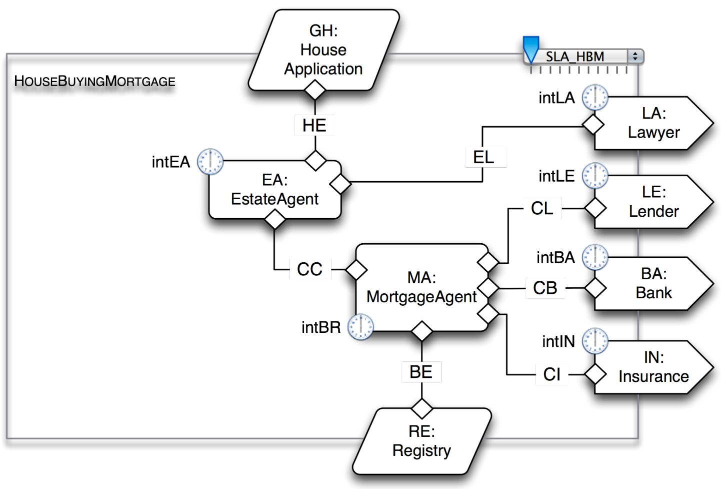


Figure 15: The result of the unification

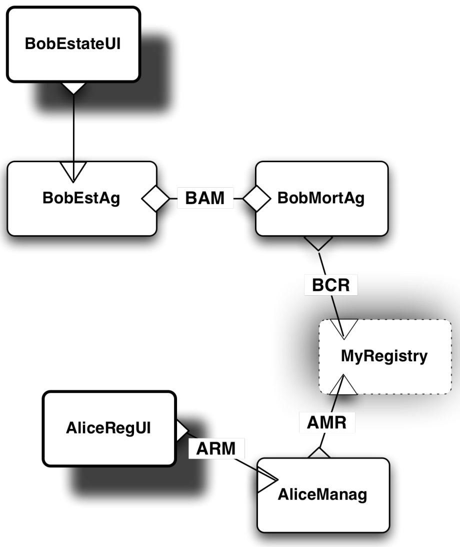


Figure 16: The result of the binding

6 Related Approaches

SRML was developed to respond to a general lack of formal support for SOC. Essentially, this new paradigm has been dominated by languages and standards developed by organisations such as OASIS (www.oasis-open.org) and W3C (www.w3.org) for Web Services. Languages such as WSDL, BPEL or BPMN address the need for developing applications over service-oriented architectures but they lack proper mathematical foundations. The fact that the modelling framework that we are building around SRML is equipped with a formal semantics makes it possible to support the analysis of services, a direction that we are pursuing through the use of model-checking techniques [4] and stochastic process algebra [9]. On the other hand, other formal approaches to SOC such as recent proposals for *service calculi* (e.g., [16,20,21,33]) have focused on the need for specialised language primitives for *programming* in this new paradigm. Such calculi can be seen to provide operational foundations for SOC (in the sense of how services compute), namely a mathematical semantics for the mechanisms that support 'choreography' or 'orchestration' – sessions, message/event correlation, compensation, inter alia. Our approach addresses instead 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.

As discussed in Section 3, SRML is based on a logic for specifying and reasoning about interactions in the conversational mode that characterises services [1,3]. The primitives that we propose take into account proposals that have been made for Web-Service Conversation [7], in other modelling languages such as ORC [36], and in a number of process calculi (e.g. [16,20,33]). More specifically, they take into account that interactions are stateful and provide first-class notions such as reply, commit, compensation and pledge, which leads to a formalism that addresses the specific needs of service-oriented modelling.

In what concerns the specification of service interfaces (also known as *service contracts* [22]*)*, we should point out that the style of specification that we adopt is quite different from recent proposals in the area of Semantic Web-Services (METEOR-S, OWL-S, SWSL, WSMF), which go little beyond a black-box, transformational approach based on concepts like pre- and post-conditions for services. These contribute to some extent towards a behavioural description of services but are confined to static/transformational aspects of black-box behaviour that only takes into account initial and final states of service execution. An exception is [42], which adopts an assumption/commitment style of specification as used for concurrent and distributed processes. Richer service interfaces found in other approaches that encompass business protocols often rely on process-oriented formalisms such as state machines [7,15,30], process calculus [16,20,21], Petri nets [34,38] or patterns (e.g., *message exchange patterns* included in WSDL 2.0). In such cases, the service description includes the message exchange sequences that are supported by the service, abstracting from the exchanged data. Hence, it is not possible to specify constraints over the values of exchanged data.

Another distinctive characteristic of SRML in what concerns service interfaces is the fact that it makes it possible to express business level properties encompassing (non-functional) aspects that can be open to negotiation. An example can be found in Section 4.2.2: the property that relates the charge applied to the base price of *GETMORTGAGE* service and the validity of the offered proposal. In fact, none of the approaches mentioned above addresses the dynamic aspects – discovery and binding – as offered by SRML.

The same holds for other approaches that have been proposed for service modelling and design such as [17,23,44,41]. For instance, the architectural framework proposed in [44], which was designed to be close to SCA [46], offers a meta-model that covers service-oriented modelling aspects such as interfaces, wires, processes and data, but none of the dynamic aspects: as in SCA, interfaces are syntactic and bindings are established at design time, whereas our interfaces are behavioural and binding occurs at run time. A more comprehensive meta-model for services is defined by the modelling language SoaML [41], which is based on the definition of new UML profiles. The definition of the external structure of services is also aligned with SCA and hence similar to what we proposed in SRML. While in SRML the external structure of a service is defined by one provides and a number of requires-interfaces, SoaML defines one service-point interface and a number of request-point interfaces. Those interfaces add to UML interfaces the definition of the partners that can initiate an interaction (by defining provided and required operations) and may specify a behaviour protocol (any UML behaviour specification can be used for this purpose). However, bindings are established at deployment time, when defining so-called "deployable participants". SENSORIA is also producing a more global approach to modelling service orchestrations in UML2 – called UML4SOA – and utilising these models for code generation (including BPEL code) [35,45] but, again, without support for service discovery and binding.

7 Concluding Remarks

We presented a formal approach for modelling service-oriented applications. This is part of an on-going effort that we are pursuing within the SENSORIA project towards a methodological and mathematical characterisation of the service-oriented computing paradigm [39]. Our 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 [14]) and policy languages (such as StPowla [13]), model-checking techniques that support qualitative analysis [4] and support for quantitative analysis through the use of stochastic process algebra [9]. A mathematical semantics is available for all aspects of the approach as partially illustrated in the paper (see [1,3,25,26,27,28] for a more comprehensive account).

SRML fills in an important gap in the way SOC is being supported, namely the fact that languages and models that have been proposed for service modelling and design do not address the higher level of abstraction that is associated with business solutions, in particular the key characteristic aspects of SOC that relate to the way those solutions are put together dynamically in reaction to the execution of business processes —run-time discovery, instantiation and binding of services. The overall methodology that we have in mind for developing software for global computers was discussed and illustrated through a fragment of the financial case study being investigated in SENSORIA. Applications of SRML in other domains can be found in [2] (telco), [3] (travel), [11] (automotive) and [25] (procurement).

Acknowledgments

We would like to thank our colleagues in the SENSORIA project for many useful discussions on the topics covered in this paper. We are also indebted to Colin Gilmore from Box Tree Mortgage Solutions (Leicester) for taking us through the mortgage business.

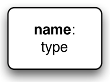
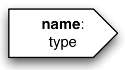
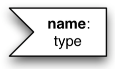
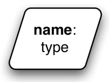
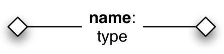
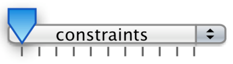
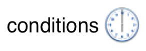
References

1. J. Abreu (2009) *Modelling Business Conversations in Service Component Architectures*. PhD thesis.
2. J. Abreu, L. Bocchi, J. L. Fiadeiro, A. Lopes (2007) Specifying and composing interaction protocols for service-oriented system modelling. In: J. Derrick, J. Vain (eds) *Formal Methods for Networked and Distributed Systems*. *LNCS, vol 4574*. Springer, pp 358–373
3. J. Abreu, J. Fiadeiro (2008) A coordination model for service-oriented interactions. In: D Lea, G. Zavattaro (eds) *Coordination Languages and Models*. *LNCS, vol 5052*. Springer, pp 1–16
4. J. Abreu, F. Mazzanti, J. Fiadeiro, S Gnesi (2009) A model-checking approach for service component architectures. In: D. Lee, A. Lopes, A. Poetzsch-Heffter *FMOODS-FORTE'09*. *LNCS, vol 5522*, Springer, 212–217
5. G. Alonso, F. Casati, H. Kuno, V. Machiraju (2004) *Web Services*. Springer
6. M. ter Beek, A. Fantechi, S. Gnesi, F. Mazzanti (2008) An action/state-based model checking approach for the analysis of communication protocols for Service-Oriented Applications. In: S. Leue, P. Merino (eds) *Formal Methods for Industrial Critical Systems*, *LNCS, vol 4916*. Springer, pp 133–148
7. B. Benatallah, F. Casati, F. Toumani (2004) Web services conversation modeling: A cornerstone for e-business automation. *IEEE Internet Computing* 8(1): 46–54
8. S. Bistarelli, U. Montanari, F. Rossi (1997) Semiring-based constraint satisfaction and optimization. *Journal of the ACM* 44(2): 201–236
9. L. Bocchi, J. Fiadeiro, S. Gilmore, J. Abreu, M. Solanki, V. Vankayala (2009) *A Formal Model for Timing Aspects of Service-Oriented Systems*. (Available from www.cs.le.ac.uk/people/jfiadeiro).
10. L. Bocchi, J. Fiadeiro, A. Lopes (2008) A use-case driven approach to formal service-oriented modelling. In: T. Margaria, B Steffen (eds) *Leveraging Applications of Formal Methods, Verification and Validation*. *CCIS, vol 17*. Springer, pp 155–169
11. L. Bocchi, J. Fiadeiro, A. Lopes (2008) Service-oriented modelling of automotive systems. In: *Proc. 32nd IEEE Int. Computer Software and Applications Conference (COMPSAC)*. IEEE, pp 1059–1064
12. L. Bocchi, J. Fiadeiro, N. Rajper, S. Reiff-Marganiec (2009) Structure and behaviour of virtual organisation breeding environments. In: J. Bryans, J. Fitzgerald (eds) *Formal Aspects of Virtual Organisations (FAVO 2009)*. University of Newcastle Technical Report (Available from www.cs.le.ac.uk/people/jfiadeiro)

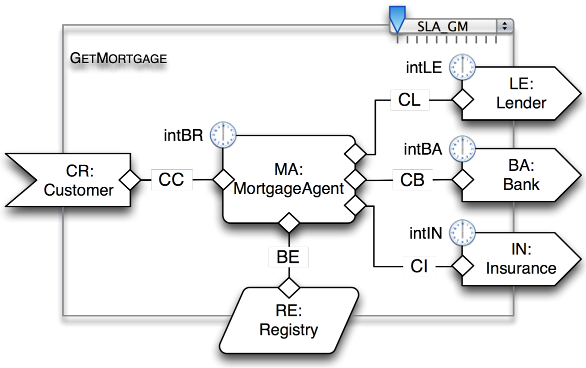
13. L. Bocchi, S. Gorton, S. Reiff-Marganiec (2008) Engineering service-oriented applications: From StPowla processes to SRML models. In: J. Fiadeiro, P. Inverardi (eds) *Fundamental Aspects of Software Engineering. LNCS, vol 4961*. Springer, pp 163–178
14. L. Bocchi, Y. Hong, A. Lopes, J. Fiadeiro (2007) From BPEL to SRML: a formal transformational approach. In: M. Dumas, R. Heckel (eds) *Web Services and Formal Methods. LNCS, vol 4937*. Springer, pp 92–107
15. L. Bordeaux et al (2005) When are two web services compatible? In: *Technologies for E-Services. LNCS, vol 3324*. Springer, pp 15–28
16. M. Boreale et al (2006) SCC: a service centered calculus. In: M. Bravetti, M. Nunez, G. Zavattaro (eds) *Web Services and Formal Methods. LNCS, vol 4184*. Springer, pp 38–57
17. M. Broy, I. Krüger, M. Meisinger (2007) A formal model of services. *ACM TOSEM* 16(1): 1–40
18. M. Broy (2007) From "Formal Methods" to System Modelling. In *Bjorner/Zhou Festschrift. LNCS, vol 4700*. Springer, pp 24–44
19. L. Camarinha-Matos, H. Afsarmanesh (2003) Elements of a base VE infrastructure. *Journal of Computers in Industry* 51(2): 139–163
20. M. Carbone, K. Honda, N. Yoshida (2007) Structured communication-centred programming for web services. In R. De Nicola (ed) *ESOP'07. LNCS, vol 4421*. Springer, pp 2–17
21. G. Castagna, N. Gesbert, L. Padovan (2009) A theory of contracts for Web services, *ACM Trans. Program. Lang. Syst.* 31(5):1–61
22. F. Curbera (2007) Component Contracts in Service-Oriented Architectures. *IEEE Computer* 40(11): 74–80
23. R. M. Dijkman and M. Dumas (2004) Service-oriented design: a multi-viewpoint approach. *International Journal of Cooperative Information Systems* 13(4): 337–368.
24. A. Elfatatry (2007) Dealing with change: components versus services. *Communications of the ACM* 50(8): 35–39
25. J. L. Fiadeiro, A. Lopes, L. Bocchi (2006) A formal approach to service-oriented architecture. In: M. Bravetti, M. Nunez, G. Zavattaro (eds) *Web Services and Formal Methods. LNCS, vol 4184*. Springer, pp 193–213
26. J. L. Fiadeiro, A. Lopes, L. Bocchi (2007) Algebraic semantics of service component modules. In: J. L. Fiadeiro, P. Y. Schobbens (eds) *Algebraic Development Techniques. LNCS, vol 4409*. Springer, pp 37–55
27. J. L. Fiadeiro, A. Lopes, L. Bocchi (2008) *An Abstract Semantics of Service Discovery and Binding*. Submitted. (Available from www.cs.le.ac.uk/people/jfiadeiro)
28. J. L. Fiadeiro, V. Schmitt (2007) Structured co-spans: an algebra of interaction protocols. In T. Mossakowski, U. Montanari, M. Haveraaen (eds) *Algebra and Coalgebra in Computer Science. LNCS, vol 4624*. Springer, pp 194–20
29. I. Foster, C. Kesselman (eds) (2004) *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann
30. H. Foster, S. Uchitel, J. Magee, J. Kramer (2006) WS-Engineer: A Tool for Model-Based Verification of Web Service Compositions and Choreography. In *ICSE 2006*, ACM Press.
31. Q. Gu, P. Lago (2007) A stakeholder-driven service life-cycle model for SOA. In *IW-SOSWE'07*. ACM Press 1–7
32. J. Hillston (1996) *A Compositional Approach to Performance Modelling*. Cambridge University Press
33. A. Lapadula, R. Pugliese, F. Tiezzi (2007) Calculus for orchestration of web services. In: R. De Nicola (ed) *ESOP'07. LNCS, vol 4421*. Springer, pp 33–47
34. A. Martens (2005). Analyzing Web Service Based Business Processes. In: M. Cerioli (ed), *FASE 2005*. LNCS, *vol. 3442*. Springer, 19–33.

35. P. Mayer, N. Koch, A. Schröder (2008) A Model-Driven Approach to Service Orchestration. In: *Proceedings of IEEE International Conference on Services Computing* (SCC 2008). IEEE Press, pp 533–536
36. J. Misra, W. Cook (2006) Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modelling* 6(1): 83–110
37. C. Peltz (2003) Web services orchestration and choreography. *IEEE Computer* 36(10):46–52
38. W. Reisig (2005) Modeling and analysis techniques for web services and business processes. In: *FMOODS 2005*, *LNCS, vol 3535*. Springer, pp 243–258
39. SENSORIA consortium (2007) White paper available at http://www.sensoria-ist.eu/files/whitePaper.pdf
40. M. Shaw, D. Garlan (1996) *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, London
41. SoaML. Service oriented architecture Modeling Language. Publications and specifications available from www.omgwiki.org/SoaML/
42. M. Solanki, A. Cau and H. Zedan (2004) Augmenting semantic web service description with compositional specification. In *WWW'04*. ACM Press, New York, pp 544–552
43. UDDI Spec TC (2004) *UDDI Specification Technical Committee Draft*. Technical report, OASIS, available at uddi.org/pubs/uddi v3.htm/
44. W. van der Aalst, M. Beisiegel, K. van Hee, D. Konig (2007) An SOA-based architecture framework. *Journal of Business Process Integration and Management* 2(2): 91–101
45. M. Wirsing, A. Clark, S. Gilmore, M. Hölzl, A. Knapp, N. Koch, A. Schröder (2006) Semantic-based development of service-oriented systems. In: E. Najn et al. (eds) *Formal Methods for Networked and Distributed Systems*. *LNCS, vol 4229*. Springer, pp 24–45
46. The Open Service Oriented Architecture collaboration. Whitepapers and specifications available from www.osoa.org (see also oasis-opencsa.org/sca)

Appendix A – The Iconography

icon	represents	type	sections
	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 Example



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

C_1: {MA.CHARGE,MA.getProposal💣}

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

C_2: {LE.ServiceId}

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

C_3: {MA.getProposal💣,LE.requestMortgage💣},

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

C_4: {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 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

MA MortgageAgent	c_1	**CB**	d_1	**BA** Bank
s&r openAccount 🔔idData loanData	S_1 i_1 i_2	Straight. I(usrdata, loandata)	R_1 i_1 i_2	**r&s** newMortgageAccount 🔔 idData loanData

✉ accountData	o_1	O(accountdata)	o_1	✉ accountData

MA MortgageAgent	c_1	**CI**	d_1	**IN** Insurance
s&r getInsurance	S_1	Straight.	R_1	**r&s** newMortgageInsurance
🔔 idData	i_1	I(usrdata,	i_1	🔔 idData
loanData	i_2	loandata)	i_2	loanData
✉ insuranceData	o_1	O(insurancedata)	o_1	✉ insuranceData

MA MortgageAgent	c_1	**CL**	d_1	**LE** Lender
s&r askProposal	S_1	Straight.	R_1	**r&s** requestMortgage
🔔 idData	i_1	I(usrdata,	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	**CC**	d_1	**MA** MortgageAgent
S_1	Straight.	R_1	**r&s** getProposal
i_1	I(usrdata,	i_1	🔔 idData
i_2	moneyvalue,prefdata)	i_2	income
i_3	O(mortageproposal,	i_3	preferences
o_1	moneyvalue)	o_1	✉ proposal
o_2		o_2	cost
R_1	Straight	S_1	**snd** confir-mation
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)
    sends askProposal🔔
       ∧ askProposal.idData=getProposal.idData
       ∧ askProposal.income=getProposal.income

 transition GetProposal
    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'=SIGNINING
       ∧ accountData=openAccount.accountData
    sends ¬needInsurance ⊃ signOutLoan🔔
                ∧ signOutLoan.insuranceData=insuranceData
                ∧ signOutLoan.accountData=accountData
```

```
transition GetInsurance
  | triggeredBy getInsurance✉
  | guardedBy s=PROPOSAL_ACCEPTED
  | effects needInsurance'=false
  |    ∧ ¬needAccount ⊃ s'=SIGNING
  |    ∧ insuranceData=getInsurance.insuranceData
  | sends ¬needAccount ⊃ signOutLoan🔔
  |              ∧ signOutLoan.insuranceData=insuranceData
  |              ∧ signOutLoan.accountData=accountData

transition Conclude
  | triggeredBy signOutLoan✉
  | guardedBy s=SIGNING
  | effects s'=FINAL
  | sends confirmation🔔
  |    ∧ confirmation.contract=signOutLoan.contract
  |    ∧ regContract(askProposal.loanData,signOutLoan.contract)
```

```
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🔔?
      requestSignOut✉.Reply after requestSignOut✉?
```

```
BUSINESS PROTOCOL Bank is
___________________________________________________
   INTERACTIONS
      r&s newMortgageAccount
      🔔 idData:usrdata,
         loanData:loandata,
      ✉ accountData:accountdata
   BEHAVIOUR
      initiallyEnabled newMortgageAccount🔔?
      newMortgageAccount.Reply after newMortgageAccount✉!
```

```
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✓? ensures confirmation🔔!

                                                END SPECIFICATIONS
```