A Use-Case Driven Approach to Formal Service-Oriented Modelling†

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

[1] Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
{bocchi,jose}@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. We put forward a use-case based approach for SRML – a formal framework that is being defined by the SENSORIA consortium for service-oriented modelling. We expand on the way SRML contributes to the engineering of software systems and we propose a number of extensions to the UML for supporting that approach. We use a mortgage brokerage scenario for illustrating our approach.

1 Introduction

This paper is about a new way of developing software, which we believe requires that we revisit the methods and techniques that software engineers have been using so far. This new approach is based on *Service-Oriented Computing* (SOC) over *Global Computers* (GC).

We view SOC as a new computing 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 programmed (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 that service-level agreements may involve context-dependent conditions, different instances of the same application may bind to different services and operate according to different SLAs resulting from different negotiations.

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

Having said this, one has to recognise that these capabilities of SOC as a paradigm are not always fully exploited by current Web/Grid-based technologies. One of the aims of the SENSORIA project [17] is to provide a framework in which the promise of SOC can be captured and used for evolving existing software technologies and engineering methodology. In this context, several formal languages and techniques are being developed that address different aspect or phases of the envisaged development process. Among these is SRML – the SENSORIA Reference Modelling Language – aimed at supporting the more abstract levels of design specification, what we call 'business modelling'. Modelling in SRML is independent of the languages in which services are programmed and the platforms in which they are deployed.

SRML provides a minimalistic textual language that has been devised in order to facilitate the definition of a mathematical semantics for its constructs and the whole process of service discovery and binding [1,10]. In this paper, we focus mainly on methodological aspects, namely on a process that can be followed to arrive at (formal) service models in SRML starting from informal (or semi-formal) specifications in notations that are typical of the UML, including use-case diagrams to capture requirements. The paper proceeds as follows. In Section 2, we provide an overview of the engineering 'architecture' and processes that we see supporting SOC in GC. In Section 3, we provide a brief overview of SRML. In Section 4, we investigate use cases as a means of deriving the structure of SRML modules. In Section 5, we consider the use of statecharts for the definition of the orchestration of services. As a running example, we will use a mortgage brokerage service.

2 Service-overlay Computers

Following the Global Computing EU initiative [12], 'global computers' are "computational infrastructures available globally and able to provide uniform services with variable guarantees for communication, co-operation and mobility, resource usage, security policies and mechanisms". The notion of 'service-overlay computer' explored by SENSORIA addresses precisely the development of highly distributed loosely coupled applications that can exploit services that are globally available.

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

The notion of 'system' itself, as it applies to software, also needs to be revised. If we take one of the accepted meanings of 'system' – *a combination of related elements organised into a complex whole* – we can see why it is not directly applicable to

SOC/GC: services get combined at run time and redefine the way they are organised as they execute; no 'whole' is given *a priori* and services do not compute within a fixed configuration of a 'universe'. In a sense, we are seeing reflected in software engineering the trend for 'globalisation' that is now driving the economy.

SOC brings to the front many aspects that have already been discussed about component-based development (CBD) [8]. 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" [6]. As an example, we can imagine that a bank will have available a collection of software components that implement core functionalities such as computing interests or charging commissions, which can be used in different products such as savings or loans.

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

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

This view is summarised in Figure 1, where:

- *Activities* correspond to applications developed according to requirements provided by a business organisation, e.g. the applications that, in a bank, implement the financial products that are made available to the public. The *activity repository* provides a means for a run-time engine to trigger such applications

when the corresponding requests are published, say when a client of the bank requests a loan at a counter or through on-line banking. Activities may be implemented over given components (for instance, a component for computing and charging interests) in a traditional CBD way, but they can also rely on services that will be procured on the fly using SOC (for instance, an insurance for protecting the customer in case he/she is temporarily prevented from re-paying the loan due to illness or job loss). Activities are typed by *activity modules*. As discussed in Section 3, these identify the components that activities need to be bound to when they are launched and the services (types) that they may require as they execute. Modules also include a specification of the workflow that orchestrates the interactions among all the parties involved in the activity.

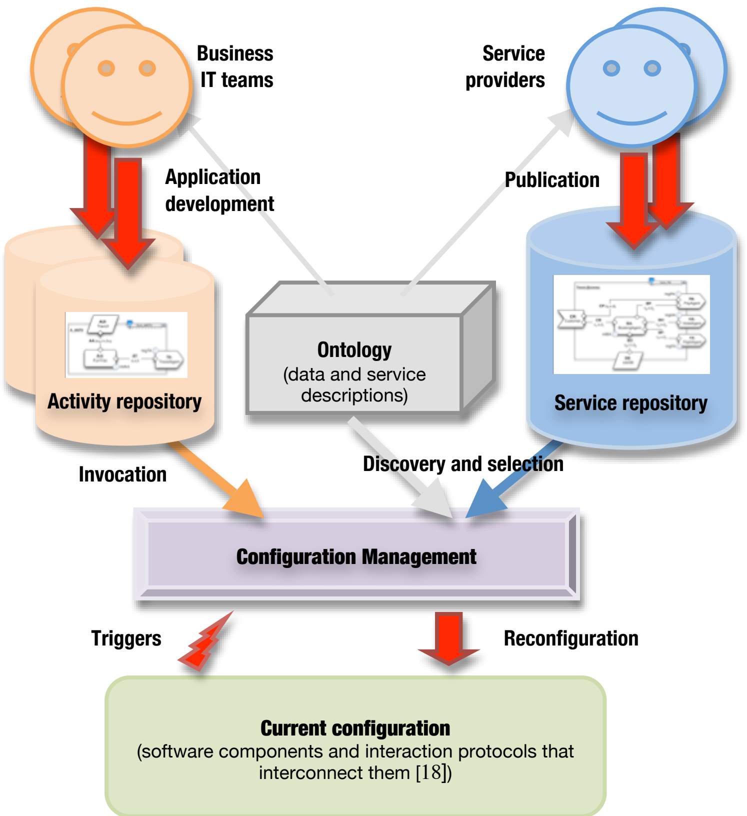


Figure 1: Overall ‘engineering’ architecture and processes

- *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 the next section we call ‘business protocols’ – that are organised in a hierarchical ontology to

facilitate discovery. Services are typed by 'service modules', which, like activity modules, identify the components and additional services that may be required together with a specification of the workflow that orchestrates the interactions among them so as to deliver the properties declared in the service description – its 'provides-interface'. Modules also specify 'service-level agreements' that need to be negotiated during matchmaking and selection.

- The *configuration management* unit is responsible for the binding of the new components and connectors that derive from the instantiation of new activities or services. A formal model of this unit can be found in [11].
- The *ontology* unit is responsible for organising both data and service descriptions. In this paper, we do not discuss the classification and retrieval mechanisms per se. See, for instance, [14,16] for some of the aspects involved when addressing such issues. Notice that the 'business IT teams' and the 'service providers' can be totally independent and unrelated: the former are interested in supporting the business of their companies or organisations, whereas the latter run their own businesses. They share the ontology component of the architecture so that they can do business together.

3 The SENSORIA Reference Modelling Language

In this section, we provide an overview of SRML focusing on the concepts needed to understand the rationale for the use-case-based approach that is proposed in Section 4. The main modelling primitive offered by SRML is called a *module*, with two specialisations – activity and service modules – in the sense discussed in Section 2.

A module M consists of:

- A graph *graph(M)*, i.e. a set *nodes(M)* of nodes and a set *edges(M)* of M where each edge e is associated with two nodes – $e{:}n \leftrightarrow m$. Edges are also called 'wire interfaces'.
- A distinguished subset of nodes $requires(M) \subseteq nodes(M)$, called 'requires-interfaces'.
- A distinguished subset of nodes $uses(M) \subseteq nodes(M)$, called 'uses-interfaces'.
- In the case of service modules, a node $provides(M) \in nodes(M)$ distinct from *requires(M)* and *uses(M),* called the 'provides-interface'.
- In the case of activity modules, a node $serves(M) \in nodes(M)$ distinct from *requires(M)* and *uses(M),* called the 'serves-interface'.
- We denote by *components(M)* the set of *nodes(M)* that are not in *provides(M)* or *serves(M)*, nor in *requires(M)* or *uses(M)* – these are called 'component interfaces'.
- A labelling function $label_M$ that assigns
  - A 'business role' to every $n \in components(M)$
  - A 'business protocol' to every $n \in provides(M) \cup requires(M)$
  - A 'layer protocol' to every $n \in serves(M) \cup uses(M)$

- A connector $<\mu_A,P,\mu_B>$ to every edge *(e:n↔m)* where *P* is an 'interaction protocol' with two 'roles' $roleA_P$ and $roleB_P$, and μ_A (resp. μ_B) is an attachment between $roleA_P$ and $label_M(n)$ (resp. $roleB_P$ and $label_M(m)$).

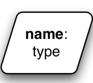
- An internal configuration policy (indicated by the symbol ◷) consisting of
  - For every node *n∈requires(M)*, a condition *trigger(n)* that identifies the trigger of the external service discovery process.
  - For *n∈components(M)*, two boolean functions *init(n)* and *term(n)* that determine initialisation and termination conditions, respectively.
- An external configuration policy (indicated by SLA) consisting of:
  - A constraint system *cs(M)* based on a fixed c-semiring [4].
  - A set *sla(M)* of constraints over *cs(M)*.
  - For every variable in *cs(M)*, a *type*.
  - A partial assignment *owner* of either a node or an edge of *M* to the variables of *cs(M)*.

Variables and constraints in *cs(M)* determine the quality profile to which the discovered services need to adhere. A precise account can be found in [11].

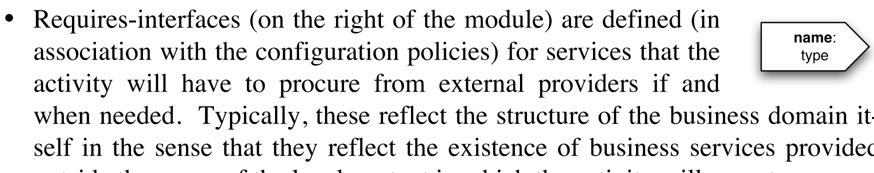
The formalisms used in SRML for defining business roles, business protocols, layer protocols and interaction protocols are discussed in [2,9]: business roles are (declarative) specifications of state transition systems in terms of state variables, triggers, guards, and publication of events; business protocols consist of temporal logic sentences (we are using a version of UCTL [3]) that specify properties of the (service-oriented) conversations held with external parties; interaction protocols are specifications of the way wires coordinate interactions between parties; layer protocols specify properties of the (component-based) interactions held with persistent components and top-level users.

An activity module (ACT), depicted using the diagrammatic notation adopted in SRML, is shown in Figure 2:

- The 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.



- 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. The corresponding layer protocols identify the views of those components that the activity will need to see supported, i.e. 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 of the module) are defined (in association with the configuration policies) 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.



- Component and wire interfaces (inside the module) should be defined for orchestrating all these entities (actors) in ways that will deliver stated user requirements through the serves-interface. The actual choice of the component interfaces and corresponding business roles may also reflect the existence of pre-defined patterns of orchestration that are available to the designers or reflect business components that will be created in support of the activity.
- The choice of the internal architecture of the module (components and wires) should also reflect the nature of the communication and distribution network over which the activity will run.

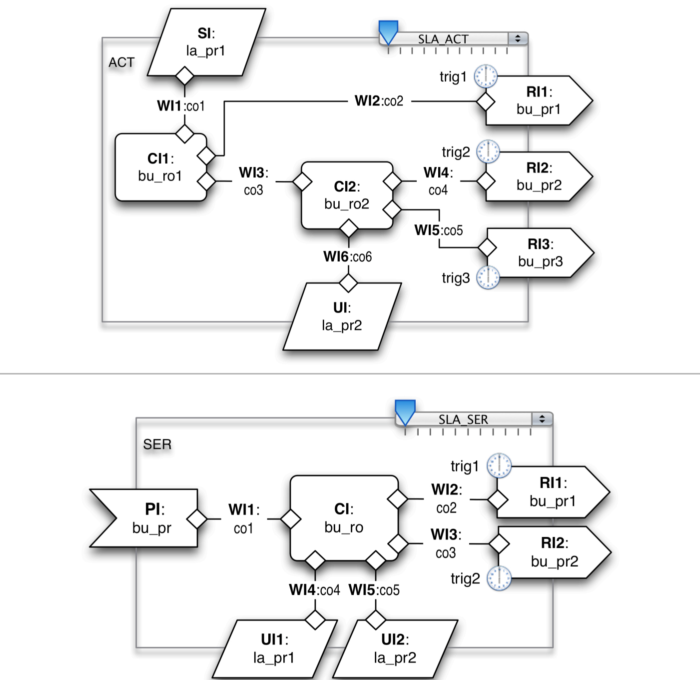
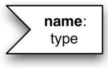


Figure 2: Diagrammatic notation for activity (top) and service (bottom) modules

In the case of a service module, a similar diagrammatic notation is used except that a provides-interface is used instead of a server-interface, as shown at the bottom of Figure 2 (module SER). In this case:

- 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.



- 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.

4 From Use-Case Diagrams to SRML

In this section, we propose an extension of use-case diagrams for service-oriented applications and discuss how to use these diagrams to obtain the skeleton of SRML modules. In order to illustrate our proposal, we will use a fragment of a financial case study. We consider the case of a financial services organisation that wants to develop a mortgage brokerage service *GetLoan* capable of binding a customer activity with a number of components with which it needs to interact to get a mortgage. This service involves the following steps: (1) proposing the best mortgage deal to the customer that invoked the service; (2) taking out the loan if the customer accepts the proposal; (3) opening a bank account associated with the loan if the lender does not provide one; and (4) getting insurance if required by either the customer or the lender.

The selection of lenders needs to be 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. Reporting to Figure 1, notice that while the aim is to publish *GetLoan* in a service repository for being discovered and invoked by other services, the *UpdateRegistry* activity 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).

4.1 Use-case diagrams for service-oriented applications

Traditionally, use-case diagrams are used for providing an overview of usage requirements for a system that needs to be built. As discussed in Section 2, 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).

The methodological implications of this view are twofold. 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. For instance, in our example, the *UpdateRegistry* activity and the *GetLoan* service can be seen to operate as part of a same business unit and, hence, it makes sense to group them together in the same use-case diagram. That is, use-case diagrams may become useful to structure usage requirements of units of business logic.

In order to reflect these methodological implications in the usage of use cases, we propose a number of extensions to the standard notation. Figure 3 illustrates our

proposal using the mortgage example: 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* service, 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 each candidate lender.

In the *GetLoan* activity, 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 of a new account. Similarly, the use case involves interaction with an *Insurance* provider for cases where the lender requires insurance or the customer decides to get one.

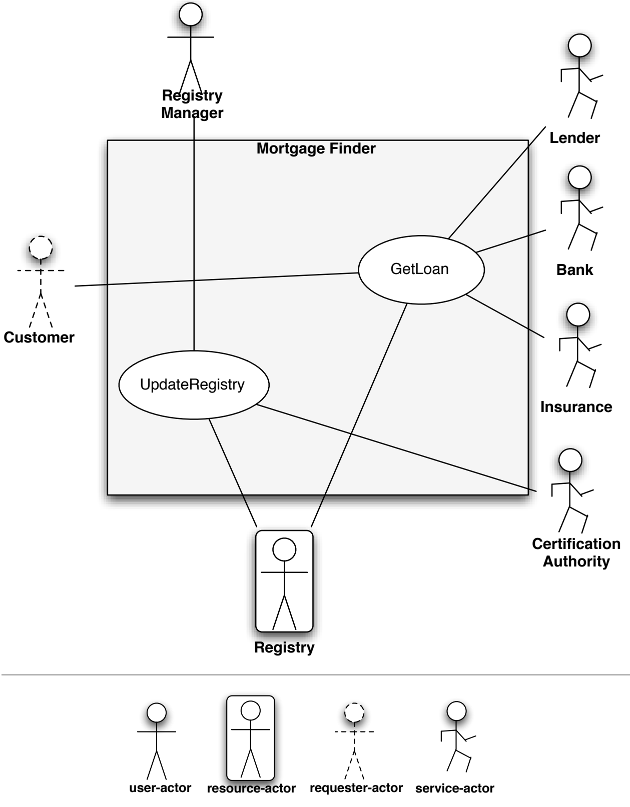


Figure 3: Service-oriented use-case diagram for *Mortgage Finder*

As happens 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 3. 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 belonging to the business organisation that operates the activity triggered by the entity, while a *requester-actor* is a role played by any entity (usually belonging to a different business organisation) that triggers the discovery of (and binds to) the service.

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

The user- and resource-actors, which we represent on 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).

4.2 Deriving the structure of SRML modules

The proposed specialisations of use-case diagrams allow us to derive a number of aspects of the structure of SRML modules. Each use case, representing either a service or an activity, is naturally modelled as either a SRML service module or activity module, respectively. The actors associated with a use case identify the interfaces used in the module. It is straightforward to model each actor type with a specific type of interface of the SRML module. Each user-actor, which represents the interface to the user that triggers the instantiation of an activity, is modelled as a SRML serves-interface. Each requester-actor, which represents the interface to the entity that invokes a service, is modelled as a SRML provides-interface. Similarly, service-actors are modelled as requires-interfaces and resource-actors as serves-interfaces. Figure 4 presents the structure of the modules derived from the use-case diagram in Figure 3.

The definition of the internal structure of the module (i.e., the components and wires that define the internal workflow) may depend on the portfolio of components already available for reuse within the business organisation. In our case study, the orchestration of the modules relies on a single component. The definition of a complex internal structure from scratch, deriving from the decomposition of the orchestration in a number of coordinated units, can be done using traditional techniques for CBD. We leave this topic for further investigation and discussion.

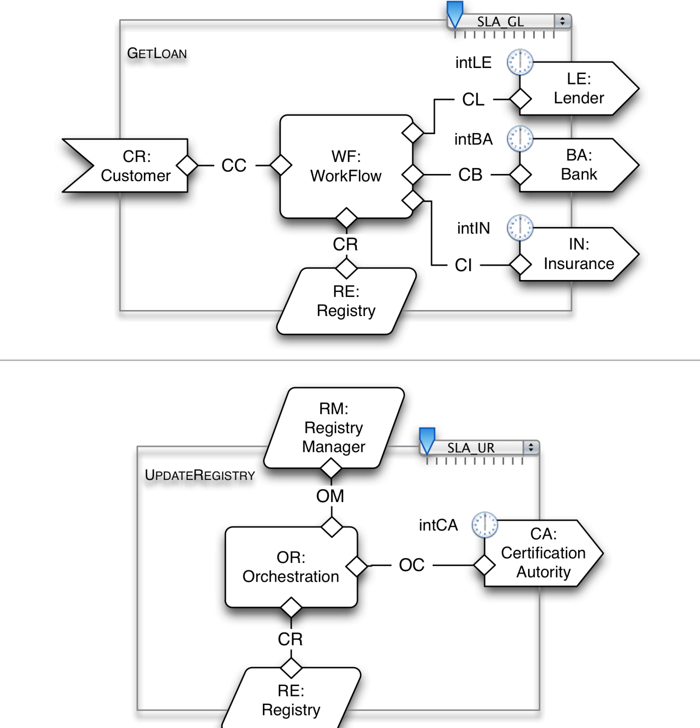


Figure 4: The SRML modules for the service *GetLoan* and the activity *UpdateRegistry*

5 Using Statecharts for SRML Orchestration

Section 4.2 describes how to derive the structure of SRML modules corresponding to use cases. In this section, we discuss how in SRML we model the internal behaviour of a module in terms of a (possibly distributed) orchestration of a number of interactions among the identified partners. For this purpose, we adapt UML statechart diagrams to operate with the interaction primitives that are available in SRML.

We illustrate the method considering the orchestration of the SRML module *GetLoan*. Initially, the customer sends his/her profile and preferences for the mortgage. If the customer accepts the proposal, and depending on the services provided by the lender, some additional activities may be performed separately: opening a bank account and buying insurance. The workflow terminates when the customer rejects the proposal or the deal is signed off.

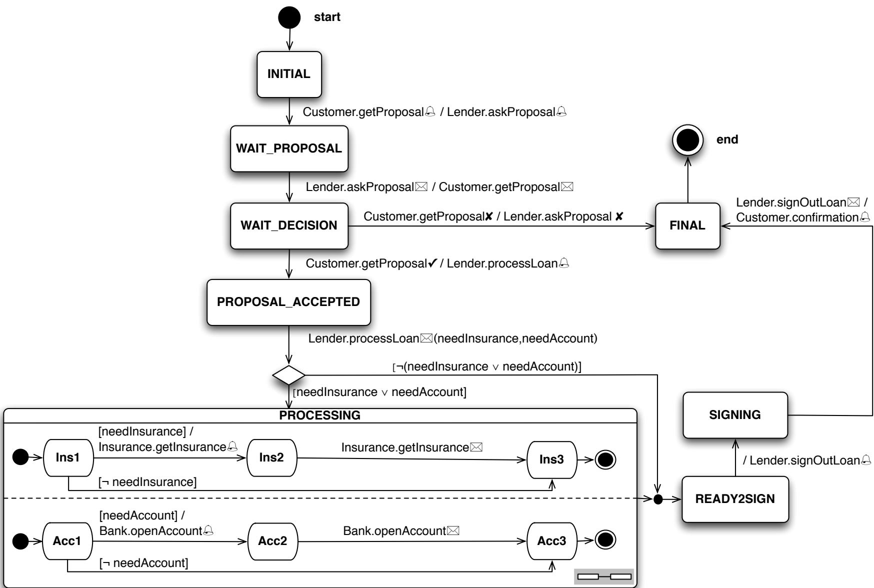


Figure 5: Statechart diagram for *GetLoan*

Figure 5 presents the statechart corresponding to the orchestration of *GetLoan*. The labels of the transitions (triggers and effects) use the language of interaction events that is provided by SRML. SRML supports asynchronous one-way (*receive* and *send*) and conversational (*send&receive* and *receive&send*) interactions. A conversational interaction may involve a number of possible steps, which we call interaction events: the initiation of the interaction (e.g., *getProposal*🔔), the reply event (e.g., *getProposal*✉) sent by the co-partner, the confirmation and cancellation events (e.g., *getProposal*✓ and *getProposal*✗), and a revoke event (e.g., *getProposal*✞) that triggers a compensation process. A one-way interaction is associated with only one event. The language and semantics of this language is discussed in [1,2].

Such a statechart defines what in Section 3 we called a 'business role', i.e. the type of orchestration that every instantiation of the service will implement to coordinate the interactions among the parties involved in the provision of the service. SRML also offers a textual notation for business roles that consists of a declaration of the interactions in which the components can be involved and a specification of the state parameters and state transitions of the orchestration process.

Figure 6 presents the interactions supported by *WorkFlow*, the specification of the component *WF* of the module *GetLoan*.

```
BUSINESS ROLE WorkFlow is
INTERACTIONS
   r&s getProposal
       🔔 idData:usrdata
          income:moneyvalue
          partnerIncome:moneyvalue
          preferences:prefdata
       ✉ proposal:mortgageproposal
   s&r askProposal
       🔔 idData:usrdata
          income:moneyvalue
          partnerIncome:moneyvalue
       ✉ proposal:mortgageproposal
   s&r processLoan
       🔔 proposal:mortgageproposal
       ✉ loanData:loandatatype
          accountIncluded:bool
          insuranceRequired:bool
   s&r getInsurance
       🔔 idData:usrdata
          loanData:loaddatatype
       ✉ insuranceData:insurancedatatype
   s&r openAccount
       🔔 idData:usrdata
          loanData:loaddatatype
       ✉ accountData:accountdatatype
   s&r signOutLoan
       🔔 loanData:loandatatype
          insuranceData:insurancedatatype
          accountData:accountdatatype
       ✉ contract:loancontract
   snd confirmation
       🔔 contract:loancontract
```

Figure 6: Interactions supported by *WF* in the SRML module *GetLoan*

Each transition in a business role is defined by: a *trigger*, typically the occurrence of an event, a *guard* enabling the transition, the *effects* over the local state and the events that are published with the corresponding parameter assignments. An extract of the transitions resulting from the statechart of *GetLoan* are presented in Figure 7.

An advantage of using the (formal) specification language of business roles over statecharts is that it supports underspecification (logical formulas are used for specifying effects and publication of events) and a refinement process that allows designers to start with loose requirements over states and transitions and add detail as more knowledge is gathered about the required behaviour. Another advantage is that is provides us a formal framework to which we can map specifications in other notations such as the ones available in workflow languages like BPEL [5]. More details and examples can be found in [9].

```
ORCHESTRATION
local s:[INITIAL, WAIT_PROPOSAL, WAIT_DECISION, PROPOSAL_ACCEPTED,
      PROCESSING, READY2SIGN, SIGN, FINAL], needAccount,needInsurance:bool
transition GetClientRequest
    triggeredBy getProposal🔔
    guardedBy s=INITIAL
    effects s'=WAIT_PROPOSAL
    sends askProposal🔔
        ∧ askProposal.idData=getProposal.idData
        ∧ askProposal.income=getProposal.income
        ∧ askProposal.partnerIncome=getProposal.partnerIncome
transition GetLenderProposal
    triggeredBy askProposal✉
    guardedBy s= WAIT_PROPOSAL
    effects s'= WAIT_DECISION
    sends getProposal✉
        ∧ getProposal.proposal=askProposal.proposal
```

Figure 7: Fragment of *Workflow* in the SRML module *GetLoan*

6 Concluding Remarks and Further Work

We presented an approach for modelling service-oriented application based on (1) use-case diagrams and statecharts in order to capture requirements on units of business logic structured in terms of services and activities, and (2) the SENSORIA Reference Modelling Language (SRML) to derive formal models of those services and activities. We proposed an extension of use-case diagrams in order to identify the relevant services and activities, and derive the structure of a SRML model for each of them. We also proposed a customisation of statechart diagrams in order to model the behaviour of the business processes executed by activities and services in terms of the basic interaction primitives available in SRML.

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) [14,19]. This approach favours the use of activity diagrams. Our choice for statecharts reflects the way we organise the behaviour of each module in terms of (internal and external) partners: the idea is that the behaviour of each partner can eventually be described by one or more statecharts, and that the behaviour of the activity/service emerges from the concurrent execution of these statecharts. This is also the view that is supporting analysis through the use of model-checking techniques [3,13]. This is on-going joint research between Leicester and ISTI (Pisa).

The overall methodology that we have in mind for developing software for global computers was also discussed and illustrated through (a much simplified version of) the financial case study being investigated in SENSORIA, namely the aspects that relate to a mortgage brokering service and registry activity. A novel aspect of SRML is the separation that it provides for services in the sense of component-based development (CBD) and service-oriented computing (SOC). This separation is reflected in the use of different kinds of actors in the proposed extension of use-case diagrams and different modelling primitives in SRML.

The specific formal support that is available in SRML was deliberately omitted because of lack of space but it can be found in a number of publications [e.g. 1,2,9,10,11]. This includes a computational model and associated logic through which we can reason about the properties of provided services using model-checking techniques [13], and also a formalism for service-level agreements [7]. However, the integrated use of these techniques within the overall methodology is still being investigated, including the support for the classification of service descriptions within an ontology that can support dynamic discovery. We are also investigating how the decomposition of use-cases using <<*include*>>/<<*extend*>> relationships, usually used to indicate potential reuse, can suggest better ways of structuring the orchestration of services and activities, as well as facilitate the checking of the properties of SRML modules.

Acknowledgments

We would like to thank our colleagues in the SENSORIA project for many useful discussions on the topics covered in this paper, especially Reiko Heckel for his insights and suggestions on use cases. We are also indebted to Colin Gilmore from Box Tree Mortgage Solutions (Leicester) for taking us through the mortgage business.

References

1. 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, Berlin Heidelberg New York, pp 1–16
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, Berlin Heidelberg New York, pp 358–373
3. 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, Berlin Heidelberg New York, pp 133–148
4. S. Bistarelli, U. Montanari, F. Rossi (1997) Semiring-based constraint satisfaction and optimization. *Journal of the ACM* 44(2): 201–236
5. L. Bocchi, Y. Hong, A. Lopes, J. Fiadeiro (2008) From BPEL to SRML: a formal transformational approach. In: M. Dumas, R. Heckel (eds) *Web Services and Formal Methods*. *LNCS, vol 4937*. Springer, Berlin Heidelberg New York, pp 92–107
6. M. Broy, I. Krüger, M. Meisinger (2007) A formal model of services. *ACM TOSEM* 16(1): 1–40
7. M. Buscemi, U. Montanari (2007) CC-Pi: A constraint-based language for specifying service level agreements. In: R. De Nicola (ed) *ESOP'07*. *LNCS, vol 4421*. Springer, Berlin Heidelberg New York, pp 18–32
8. A. Elfatatry (2007) Dealing with change: components versus services. *Communications of the ACM* 50(8): 35–39
9. 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, Berlin Heidelberg New York, pp 193–213
10. 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, Berlin Heidelberg New York, pp 37–55
11. J. L. Fiadeiro, A. Lopes, L. Bocchi (2008) *Semantics of Service-Oriented System Configuration*. Submitted. (Available from www.cs.le.ac.uk/jfiadeiro)
12. Global Computing Initiative, *http://cordis.europa.eu/ist/fet/gc.htm*
13. S. Gnesi, F. Mazzanti (2004) On the fly model checking of communicating UML state machines. In *ACIS International Conference on Software Engineering Research, Management and Applications*, pp 331–338

14. 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, in print
15. K. Pahl (2007) An ontology for software component matching. *International Journal on Software Tools and Technology Transfer* 9: 169–178
16. J. Rao, X. Su (2004) A survey of automated web service composition methods. In: J. Cardoso, A. Sheth (eds) *Semantic Web Services and Web Process Composition. LNCS, vol 3387*. Springer, Berlin Heidelberg New York, pp 43–54
17. SENSORIA consortium (2007) White paper available at http://www.sensoria-ist.eu/files/whitePaper.pdf
18. M. Shaw, D. Garlan (1996) *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, London
19. M. Wirsing, A. Clark, A. 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, Berlin Heidelberg New York, pp 24–45