A Model for Dynamic Reconfiguration in Service-oriented Architectures

José Luiz Fiadeiro[1] and Antónia Lopes[2]

[1]Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
jose@mcs.le.ac.uk
[2]Faculty of Sciences, University of Lisbon
Campo Grande, 1749–016 Lisboa, Portugal
mal@di.fc.ul.pt

Abstract. The importance of modelling the dynamic architectural characteristics of software systems has long been recognised. However, the nature of the dynamic architectural characteristics of service-oriented applications goes beyond what is currently addressed by existing architecture description languages (ADLs). At the heart of the service-oriented approach is the logical separation of *service need* from the need-fulfillment mechanism, i.e., the *service provider*: the binding between the two is deferred to runtime and established at the instance level, i.e. each time the need for the service emerges. In this paper we present an architecture-oriented model for dynamic reconfiguration that paves the way for the definition of ADLs that are able to address the specification of dynamic architectural characteristics of service-oriented applications.

1 Introduction

Several architectural aspects arise from service-oriented computing (SOC), loosely understood as a paradigm that supports the construction of complex software-intensive systems from entities, called services, that can be dynamically (i.e. at run time) discovered and bound to applications to fulfil given business goals. On the one hand, we have so-called service-oriented architecture (SOA), normally understood as a (partially) layered architecture in which business processes can be structured as choreographies of services and services are orchestrations of enterprise components. SOAs are supported by an integration middleware providing the communication protocols, brokers, identification/binding/composition mechanisms, and other architectural components that support a new architectural style. This style is characterised by an interaction model between service consumers and providers that is mediated by brokers that maintain registries of service descriptions and are capable of binding the requester who invoked the service to an implementation of the service description made available by a provider that is able to enter into a service-level agreement (SLA) with the consumer.

On the other hand, this new style and form of enterprise-scale IT architecture has a number of implications on the nature of the configurations (or run-time architectures) of the systems that adhere to that style (what we will call service-oriented systems). If we take one of the traditional concepts of architecture as being "concerned with

the selection of architectural elements, their interactions and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design" [30], it is possible to see why service-oriented systems fall outside the realm of the languages and models that we have been using so far for architectural description: for service-oriented systems, the selection of their architectural elements (components and connectors) is not made at design time; as new services are bound, at run time, to the applications that, in the system, trigger their discovery, new architectural elements are added to the system that could not have been anticipated at design time. In other words, the new style is essentially 'dynamic' in the sense that it applies not only to the way configurations are organised but, primarily, to the way they evolve.

For example, a typical business system may rely on an external service to supply goods; in order to take advantage of the best deal available at the time the goods are needed, the system may resort to different suppliers at different times. Each of those suppliers may in turn rely on services that they will need to procure. For instance, some suppliers may have their own delivery system but others may prefer to outsource the delivery of the goods; some delivery companies may have their own transport system but prefer to use an external company to provide the drivers; and so on. In summary, the structure of a service-oriented system, understood as the components and connectors that determine its configuration, is intrinsically dynamic. Therefore, the role of architecture in the construction of a service-oriented system needs to go beyond that of identifying, at design time, components and connectors that developers will need to implement. Because these activities are now performed by the SOA middleware, what is required from software architects is that they identify and model the high-level business activities and the dependencies that they have on external services to fulfil their goals.

Run-time architectural change is itself an area of software engineering that has deserved a lot of attention from the research community [3,19,26,27,29,32], mainly as a response to the need for mechanisms for enhancing adaptability and evolvability of systems in the face of changing requirements or operating conditions. Although the dynamic nature of the architecture of service-oriented systems could be thought to fall within this general remit, there are a number of specificities that suggest that a more focused and fundamental study of dynamic reconfiguration in SOA is needed. Indeed, dynamic reconfiguration is clearly intrinsic to the computational model of SOC, i.e. it is not a process that, like adaptability or evolvability, is driven by factors that are external to the system. Naturally, self-adaptation is a key concern for many systems but, essentially, this means reacting to changes perceived in the environment in which the system operates. In the case of services, the driver for dynamic reconfiguration (through change of the source of provision each time a service is required) is not so much the need to adjust the behaviour in response to changes in the environment: it is part of the way systems should be designed to meet goals that are endogenous to the business activities that they perform. In both cases, the aim is to optimise the way quality-of-service requirements are met. However, while in architectural-based approaches to self-adaptation the optimisation process is programmed in terms of reconfiguration actions, in the case of services the optimisation process is determined by quality-of-service requirements that derive from business goals.

Our purpose in this paper is to put forward a mathematical model that can be used as a semantic domain for service-oriented architectural description languages. Our starting point is the graph-based approach that we and other authors have used for architectural reconfiguration [12,32]. Essentially, we introduce a mechanism of reflection (as used in other approaches to dynamic reconfiguration [14,21]) by which configurations are typed with models of business activities and service models define rules for dynamic reconfiguration. This mathematical model was used in the SENSORIA project to define the dynamic semantics of the language SRML [18]. A full definition of the model itself cannot be provided here; a more detailed account can be found in [17]. For illustrating our approach, we use the financial case study developed in SENSORIA.

The paper is organised as follows. In Section 3, we define a model for business-reflective configurations of systems. In Section 4, we put forward a model of services as rules for the dynamic reconfiguration of systems and we outline an operational semantics for the rules defined by services. We discuss related work in Section 5 and conclude in Section 6 by pointing to other aspects that are being investigated.

2 Motivation and example

At a certain level of abstraction, configurations of service-oriented applications can be seen to be a particular case of component-connector architectural configurations: a graph of *components* (applications deployed over a given execution platform) linked through *wires* (interconnections between components over a given network)[1]. We denote by **COMP** and **WIRE** the universes of components and wires, respectively.

As it often happens in the presence of dynamic reconfiguration, it is necessary to consider the execution state of the configuration elements as well. Every component $c \in$ **COMP** and wire $w \in$ **WIRE** of a configuration may be in a number of states (e.g. valuations of local state variables), the set of which is denoted by $\mathbf{STATE}_c$ and $\mathbf{STATE}_w$, respectively. We denote by **STATE** the corresponding indexed family of sets of states.

Definition 1 (Configuration and State Configuration).

- *A* configuration *is a simple graph $\mathcal{G}$ such that $nodes(\mathcal{G}) \subseteq$* **COMP** *(i.e. nodes are components) and $edges(\mathcal{G}) \subseteq$* **WIRE** *(i.e. edges are wires). Each edge e is associated with a (unordered) pair of nodes that we denote by $e : n \leftrightarrow m$.*
- *A* state configuration *$\mathcal{F}$ is a pair $\langle \mathcal{G}, \mathcal{S} \rangle$, where $\mathcal{G}$ is a configuration and $\mathcal{S}$ is a configuration state, i.e., a mapping that assigns an element of* $\mathbf{STATE}_c$ *to each $c \in nodes(\mathcal{G})$ and an element of* $\mathbf{STATE}_w$ *to each $w \in edges(\mathcal{G})$.*

Configurations of service-oriented applications change as a result of the creation of new business activities and the execution of existing ones: new components or wires may be added to a configuration because the execution of a business activity triggered the discovery of and binding to a service that is required. In order to illustrate our approach, we use a (simplified) scenario in which there is a financial services organisation

[1] In SOC, message exchanges are essentially peer-to-peer and, hence, for simplicity, we take all connectors to be binary.

that offers a mortgage-brokerage service MORTGAGEFINDER that, in addition to finding the best mortgage deal for a mortgage request, opens a bank account associated with a loan (if the lender does not provide one) and procures an insurance policy (if required by either the customer or the lender). The provision of this service depends on three other services — a $Lender$, a $Bank$, an $Insurance$ — that are assumed to be provided by other organisations and procured at run time, each time they are needed, according to the profile of the customer and market availability.

In this context, let us consider a situation in which there is a business activity A_{Bob} processing a mortgage request issued through a user interface $BobHouseUI$ on behalf of a customer (Bob), and that this activity is being served by MORTGAGEFINDER. Suppose that the active computational ensemble of components that collectively pursue the business goal of this activity in the current state is as highlighted (through a dotted line) on the left-hand side of Figure 1 — the component $BobMortAg$ is orchestrating the delivery of MORTGAGEFINDER, which requires it to interact with the component $BobEstAg$ that is acting on behalf of Bob (who is using the interface $BobHouseUI$), and a database $MortRegistry$ of trusted lenders. Other components may be present in the current configuration that account for other business activities running in parallel with A_{Bob}, say activities processing other mortgage requests that share the same database $MortRegistry$ or, as depicted in Figure 1, updating that registry with new lenders. That is, A_{Bob} is in fact a sub-configuration of a larger system.

Let us further imagine that the discovery of a provider of the service $Lender$ is triggered by $BobMortAg$. As illustrated in the right-hand side of Figure 1, as a result of the execution of the discovery and binding process, a new component — $RockLoans$ — is added to the current configuration and bound to the component $BobMortAg$ that is

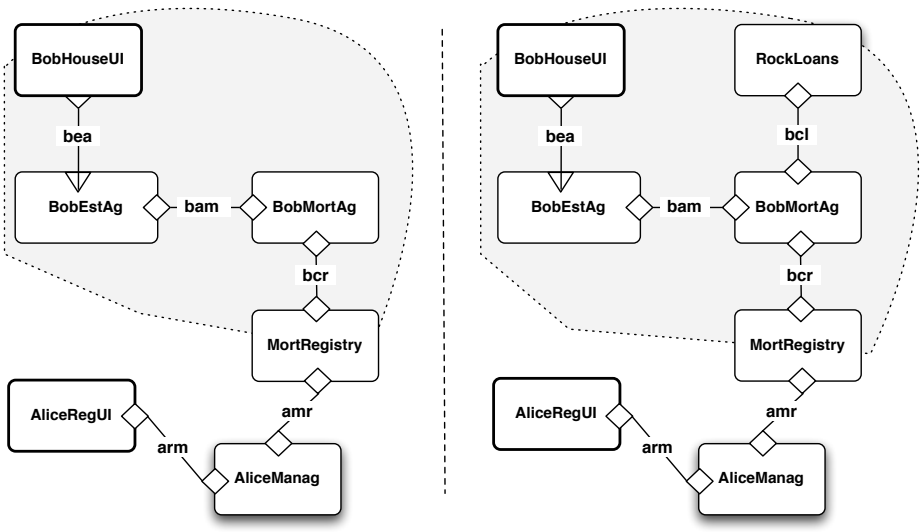

Fig. 1. Two configurations that shows the sub-configuration that corresponds to the business activity A_{Bob} before and after the discovery of a provider of the service $Lender$, respectively.

orchestrating the delivery of MORTGAGEFINDER. This new component is responsible for the provision of the service by the selected provider of $Lender$.

This example illustrates why, in order to capture the dynamic aspects of SOC, we need to look beyond the information available in a state — configurations account only for which components are active and how they are interconnected, not why they are active and interconnected in that way. Therefore, we need to have available information that accounts for the dependencies that the activity has on externally provided services, the situations in which they need to be discovered, and the criteria according to which they should be selected. The approach that we developed achieves this by making configurations *business reflective*, i.e. by labelling each sub-configuration that corresponds to a business activity with a model of the workflow that implements its business logic. The models that we propose for this effect are called *activity modules*, whose operational semantics defines the rules according to which service-oriented systems are dynamically reconfigured. We discuss this form of reflection in Section 3.

3 Business-reflective configurations

Activity modules are specification artefacts that we use for typing the sub-configurations that, in a given state, execute the business activities that are running. Figure 2 depicts the activity module that types the configuration of the activity A_{Bob} on the left-hand side of Figure 1, i.e. before the discovery of a provider of the service *Lender*. The different elements of an activity module are:

- **Component-interfaces**: the specifications that type the components that, in the sub-configuration, execute the business activity. For example, MA is a component-interface declared to be of type $MortgageAgent$.
- **Serves-interface**: the specification of the interface (HUI in the example) that the activity uses to interact with users.
- **Uses-interfaces**: the specification of the interactions that the activity performs with persistent components (MR of type $Registry$ in the example).
- **Wire-interfaces**: the connectors — roles and glue, in the sense of [4] — that specify, through the glue, the protocols that are executed by the wires and the maps from the roles of the connectors to the component specifications.
- **Requires-interfaces**: the specifications of the external services that may be required during the execution of the activity. For instance, the activity module in Figure 2 declares three 'requires-interfaces' — LA of type $Lawyer$, IN of type $Insurance$, LE of type $Lender$ and BA of type $Bank$. These types are specifications of the behaviour that is required of external services. They are used for the selection of providers when the discovery of the services is triggered.
- **Internal configuration policies**: these are state conditions associated with component interfaces that specify how they should be initialised, and triggers associated with requires-interfaces that determine when external services need to be discovered. Graphically, these policies are identified by the clocks.
- **External configuration policies**: these are the SLA constraints that apply to the discovery and selection of external services. Graphically, these policies are identified by the rulers.

The nature of the specifications used for defining the interfaces is not relevant for the purpose of this paper. In [18] we have used both a declarative language and an extension of UML statecharts for component-interfaces, and temporal logic for requires-interfaces, but other formalisms could be used. For generality, we assume that all specifications belong to a universe **SPEC**. We distinguish between the different kinds of interfaces because they have different roles in the dynamic re-configuration of the activity as explained further on. We also abstract from the nature of the connectors that are used in wire-interfaces and work over a generic universe **CNCT**. Details on the kind of connectors that we have found useful for service modelling can be found in [1].

The specific language used for specifying initialisation conditions and triggers is also of no particular importance for this paper, so we assume that we have available a set **STC** of conditions over **STATE**. Finally, we adopt so called 'soft constraints' for expressing SLA constraints. These generalise the notion of constraint: while a constraint is a predicate over a certain set of variables X and, hence, divides the set of valuations of X in two disjoint subsets (those that satisfy the constraint and those that do not), a soft constraint is a function mapping each valuation of X into some domain D (e.g., the interval of real numbers $[0, 1]$) that captures different degrees of satisfaction. Soft constraints are commonly used for describing problems where it is necessary to model fuzziness, preferences, costs, inter alia. In particular, they have shown to be useful for supporting the negotiation of service-level agreements [7]. Some well-known soft constraint formalisms are *Valued Constraint Satisfaction Problems* [16] and *Semiring-based Soft Constraints* [6]. The particular formalism that is adopted is not relevant for this paper; in SRML [18], we adopted [6].

In summary, an activity module includes all the information that defines the business aspect of the activity on a particular state. This includes the specifications of the components and connectors that execute the activity on that state but also the dependencies on

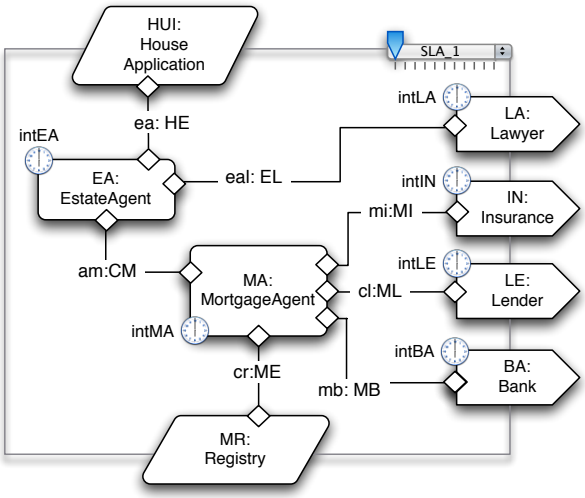

Fig. 2. The activity module that types the sub-configuration that corresponds to A_{Bob} as shown on the left-hand side of Figure 1.

external services that determine how that configuration may change. Activity modules are also formalised as graphs:

Definition 2 (Activity Module). *An* activity module M *consists of*

- *A simple graph* $graph(M)$*; we use* $nodes(M)$ *to denote the set of its nodes.*
- *A set* $requires(M) \subseteq nodes(M)$*.*
- *A set* $uses(M) \subseteq nodes(M) \setminus requires(M)$*.*
- *A node* $serves(M) \in nodes(M) \setminus (requires(M) \cup uses(M))$*.*
  We use $components(M)$ *to denote the set of all remaining nodes.*
- *A labelling function* $label_M$ *such that*
  - $label_M(n) \in$ **SPEC** *for every node* n*.*
  - $label_M(e : n \leftrightarrow m) \in$ **CNCT** *for every edge* e*.*
- *A pair* $intPlc(M)$ *of mappings* $\langle trigger_M, init_M \rangle$ *such that* $trigger_M$ *assigns a condition in* **STC** *to each* $n \in requires(M)$ *and* $init_M$ *assigns a condition in* **STC** *to each* $n \in components(M)$*.*
- *A pair* $extPlc(M)$ *consisting of a soft constraint system* $cs(M)$ *and a set* $sla(M)$ *of soft constraints over* $cs(M)$*.*

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 also formalise the typing of state configurations with activity modules motivated before, which makes configurations business-reflective. We consider a space $\mathcal{A}$ of business activities to be given, which can be seen to consist of reference numbers (or some other kind of identifier) such as the ones that organisations automatically assign when a service request arrives.

Definition 3 (Business Configuration). *A* business configuration *is a triple* $\langle \mathcal{F}, \mathcal{B}, \mathcal{C} \rangle$ *where*

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

A homomorphism of graphs is just a mapping of nodes to nodes and edges to edges that preserves the end-points of the edges. Therefore, the homomorphism $\mathcal{C}$ of a business configuration types the nodes (components) of $\mathcal{F}(a)$ with specifications of the roles that they play in the activity — i.e. $\mathcal{C}(a)(n) : label_{\mathcal{B}(a)}(n)$ for every node n — and the edges (wires) with connectors — i.e. $\mathcal{C}(a)(e) : label_{\mathcal{B}(a)}(e)$ for every edge e.

In Figure 3, we represent a business configuration for the state configuration depicted on the left-hand side of Figure 1. For simplicity, we only show the node mappings of the homomorphisms. In addition to the business activity A_{Bob} that we have been discussing, Figure 3 reveals another business activity — A_{Alice} — in which the registry of trusted lenders $MortRegistry$ is also involved. The activity module that

types A_{Alice} defines that the business goal of this activity is to update the registry with new lenders; in the particular state being depicted, this activity still requires an external service to be discovered that can certify the new lender.

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

In summary, the homomorphism makes state configurations reflective in the sense of [14] as it adds meta (business) information to the state configuration. This information is used for deciding how the configuration will evolve (namely, how it will react to events that trigger the discovery process). Indeed, reflection has been advocated as a means of making systems adaptable through reconfiguration, which is similar to the mechanisms through which activities evolve in our model. The reconfiguration process, as driven by services, is discussed in the next section.

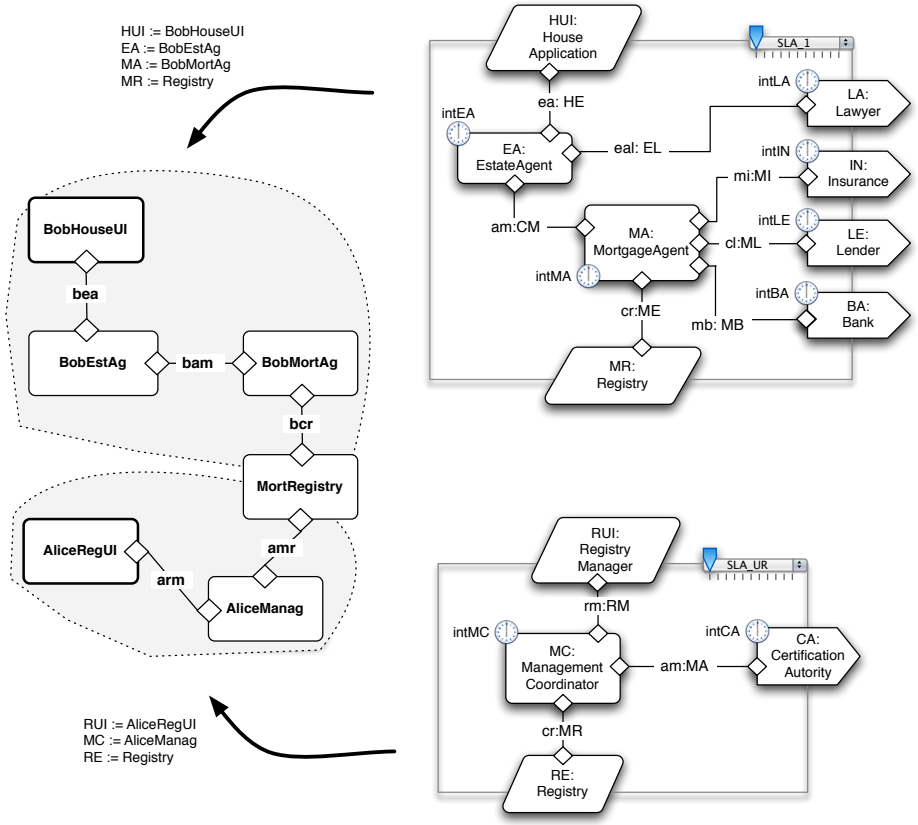

Fig. 3. A business conguration that shows the sub-congurations that correspond to the business activities A_{Bob} (top part) and A_{Alice} (bottom part) and the activity modules that type them.

4 Service Binding as a Reconfiguration Action

As already mentioned, business configurations change whenever the execution of an activity requires the discovery of and binding to a service. It remains to formally define this process, which starts with the discovery of potential providers of the service and the selection of one service provider among these.

We start by providing a formal notion of service, which we developed in SENSORIA [18] inspired by concepts proposed in the Service Component Architecture (SCA) [25]. We model services through *service modules*, which are similar to the activity modules that we introduced in the previous section except that, instead of a serves-interface to the user of the activity, they include a *provides-interface* through which activities can connect to the service (identified through a requires-interface). Such interfaces are labelled by specifications (business protocols) that describe the properties that a customer can expect from the interactions with the service. Uses-interfaces and requires-interfaces can be included in service modules in the same way as in activity modules.

Definition 4 (Service Module). *A* service module M *consists of*

- *A simple graph* $graph(M)$.
- *A set* $requires(M) \subseteq nodes(M)$.
- *A set* $uses(M) \subseteq nodes(M) \setminus requires(M)$.
- *A node* $provides(M) \in nodes(M) \setminus (requires(M) \cup uses(M))$.
- *A labelling function* $label_M$ *such that*
  - $label_M(n) \in$ **SPEC** *for every node* n.
  - $label_M(e : n \leftrightarrow m) \in$ **CNCT** *for every edge* e.
- *An internal configuration policy* $intPlc(M)$ *as in definition 2.*
- *An external configuration policy* $extPlc(M)$ *as in definition 2.*

In Figure 4 we present the structure of the service module that models the mortgage-brokerage service MORTGAGEFINDER described before. A complete definition of this service using the modelling language SRML, including all the specifications involved, is presented in [18]. The module specifies that the service is provided through an interface CR and wire CC that can bind to any activity that requests an external service through a requires-interface that is matched by the specification $Customer$. The orchestration of the provision of the service is specified through the component-interface MA of type $MortgageAgent$ which may require external services that match the requires-interfaces LE of type $Lender$ (for securing a loan), BA of type $Bank$ (for opening a bank account), and IN of type $Insurance$ (for procuring an insurance). The orchestration also requires the binding to a persistent component RE of type $Registry$ (that stores information about trusted lenders).

In order to formalise the processes of discovery and binding, let r be a requires-interface of an activity a. The discovery of services to which r can be bound involves finding services M that (i) through their provides-interface p are able to satisfy the specification associated with r, and (ii) through their external configuration policies offer SLA constraints that are compatible with those of a and, therefore, make it possible to reach a service-level agreement. For simplicity, we limit our attention to service

modules where there is exactly one component-interface connected to the provides-interface and to activity modules where each requires-interface is connected to a single component-interface (the formulation of the general case can be found in [17]).

For the formulation of condition (i) above we assume that the universe **SPEC** of specifications is equipped with a notion of *refinement* such that $\rho : r \rightarrow p$ means that the behavioural properties offered by p entail the properties required by r, up to a suitable translation between the languages of both. For example, if using temporal logic for specifying the business protocols associated with r and p as in [18] refinement corresponds to entailment (logical consequence).

The formulation of condition (ii) above relies on a composition operator $\oplus$ that is applicable to soft constraint systems that are compatible (see [6] for an example) and to sets of constraints over compatible constraints systems. Soft constraint systems also provide a notion of *best level of consistency* that assigns a non-negative numerical value $blevel(C)$ to each set of constraints C — the degree of satisfaction that we can expect for C. A set of constraints is said to be *consistent* if and only if $blevel(C) > 0$. If C is consistent, a valuation for the variables of C is said to be a *solution* of C.

Definition 5 (Service matching). *Let A be an activity module and $r \in requires(A)$. We denote by $\mathbf{match}(\mathbf{A}, \mathbf{r})$ the set of pairs $\langle M, \rho \rangle$ such that:*

- *M is a service module such that the constraint systems $cs(M)$ and $cs(A)$ are compatible and $blevel(sla(M) \oplus sla(A)) > 0$;*
- *ρ is a refinement mapping from $label_A(r)$ to $label_M(provides(M))$.*

That is, the matching process for an activity module and one of its requires-interfaces returns all service modules whose provides-interface refines the requires-interface of the activity module and whose constraint systems are compatible and whose constraints are consistent.

Definition 6 (Service Discovery). *Let A be an activity module and $r \in requires(A)$. We denote by $\mathbf{discover}(\mathbf{A}, \mathbf{r})$ the set of triples $\langle M, \rho, \Delta \rangle$ such that:*

- *$\langle M, \rho \rangle \in \mathbf{match}(\mathbf{A}, \mathbf{r})$;*

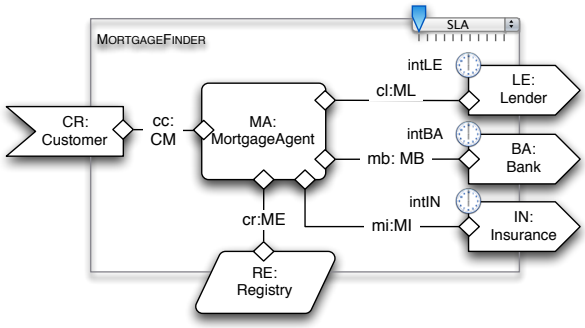

Fig. 4. The structure of a service module that models MORTGAGEFINDER.

– *Δ is a solution for $sla(M) \oplus sla(A)$ such that $bvalue(sla(M) \oplus sla(A))$ is maximal for $\mathbf{match}(\mathbf{A}, \mathbf{r})$, i.e. Δ maximises the degree of satisfaction for the combined set of SLA constraints.*

That is, the discovery process returns the set of service modules that offer the best possible service available, the solution Δ being the corresponding SLA agreement.

Consider now a business configuration $\mathcal{L} = \langle\langle \mathcal{G}, \mathcal{S}\rangle, \mathcal{B}, \mathcal{C}\rangle$, a an active business activity in $\mathcal{L}$ and $r \in requires(\mathcal{B}(a))$ such that $trigger_{\mathcal{B}(a)}(r)$ evaluates to *true* in $\mathcal{S}$. The reaction to this trigger is a reconfiguration of the business configuration, which results in a new business configuration obtained by binding an element $\langle M, \rho, \Delta\rangle$ of $\mathbf{discover}(\mathcal{B}(\mathbf{a}), \mathbf{r})$ to a. We now define this binding process.

Definition 7 (Service Binding). *Let $\mathcal{L} = \langle\langle \mathcal{G}, \mathcal{S}\rangle, \mathcal{B}, \mathcal{C}\rangle$ be a business configuration, a an active business activity in $\mathcal{L}$, $r \in requires(\mathcal{B}(a))$, M a service module, ρ a refinement mapping from r to $provides(M)$ and Δ a constraint. Binding $\langle M, \rho, \Delta\rangle$ to r induces a business configuration $\langle\langle \mathcal{G}', \mathcal{S}'\rangle, \mathcal{B}', \mathcal{C}'\rangle$ such that:*

– *$\mathcal{B}'(x) = \mathcal{B}(x)$, if $x \neq a$.*
– *$\mathcal{B}'(a)$ is an activity module M' such that:*
  - *$graph(M')$ is obtained from the sum (disjoint union) of the graphs of $\mathcal{B}(a)$ and M by identifying r with the node of M to which $provides(M)$ is connected and identifying the corresponding edges.*
  - *$requires(M') = requires(M) \cup requires(\mathcal{B}(a)) \setminus \{r\}$, i.e. we eliminate r and add the requires-interfaces of M.*
  - *$uses(M') = uses(M) \cup uses(\mathcal{B}(a))$, i.e. we add to $\mathcal{B}(a)$ the uses-interfaces of M.*
  - *$serves(M') = serves(M)$, i.e. we keep the serves-interface.*
  - *the labels provided by $label'_M$ are those that are inherited from the graphs of $\mathcal{B}(a)$ and M. The edge that connected r is now labelled with the label of the edge that connects $provides(M)$ in M.*
  - *$intPlc(M')$ has the triggers and initialisation conditions that are inherited from $\mathcal{B}(a)$ and M.*
  - *$extPlc(M') = \langle cs(M) \oplus cs(\mathcal{B}(a)), sla(M) \oplus sla(\mathcal{B}(a)) \cup \{\Delta\}\rangle$.*
– *$\mathcal{G}'$ is obtained from $\mathcal{G}$ by adding:*
  - *For each node n of $components(M)$, a component c_n in* **COMP** *that implements the specification $label_M(n)$ and, for each edge connecting n, a wire that implements the connector that labels the edge.*
  - *For every node n of $uses(M)$, a component c_n of $\mathcal{G}$ that implements the specification $label_M(n)$ is selected and, for every edge connecting n in M, a wire that implements the connector that labels the edge is added to $\mathcal{G}$.*

  That is to say, implementations of component-interfaces of M are added to the graph and existing components are chosen for uses-interfaces. Wires are added that implement the connectors specified in M.
– *$\mathcal{S}'$ coincides with $\mathcal{S}$ in the nodes of $\mathcal{G}$ and assigns, to every new node c_n where $n \in components(M)$, a state that satisfies $init_M(n)$.*
– *$\mathcal{C}'$ is the homomorphism that results from updating $\mathcal{C}$ with the mappings defined above, i.e. for each node n of $body(M)$, $\mathcal{C}'(n) = c_n$, and similarly for the edges.*

In order to illustrate how binding works, consider the business configuration in Figure 4, which shows A_{Bob} at an earlier stage of execution (i.e. earlier than the configuration depicted in the left-hand side of Figure 1). Assume that, in the current state, the trigger $intMG$ is true and that the service module shown in Figure 4 is returned by the discovery process described in Definition 6 for the requires-interface MG. A possible result of the binding is depicted in Figure 3.

Note that a new component — $BobMortAg$ — is added to the configuration of A_{Bob} as an instance of $MortgageAgent$, but that the uses-interface RE of MORTGAGEFINDER does not give rise to a new component: it is mapped to $MortRegistry$. This is the means through which effects of services can be made 'persistent', i.e. the execution of the service can interfere with other activities in the current configuration. For instance, if A_{Alice} registers a new lender, A_{Bob} will be able to consider that lender when discovering an external service that responds to the trigger $intLE$ of the requires-interface LE of type $Lender$. On the other hand, the serves-interface of the activity module remains invariant through the evolution of the business configuration. This captures the fact that the activity relies on the same interface to interact with its user. Also

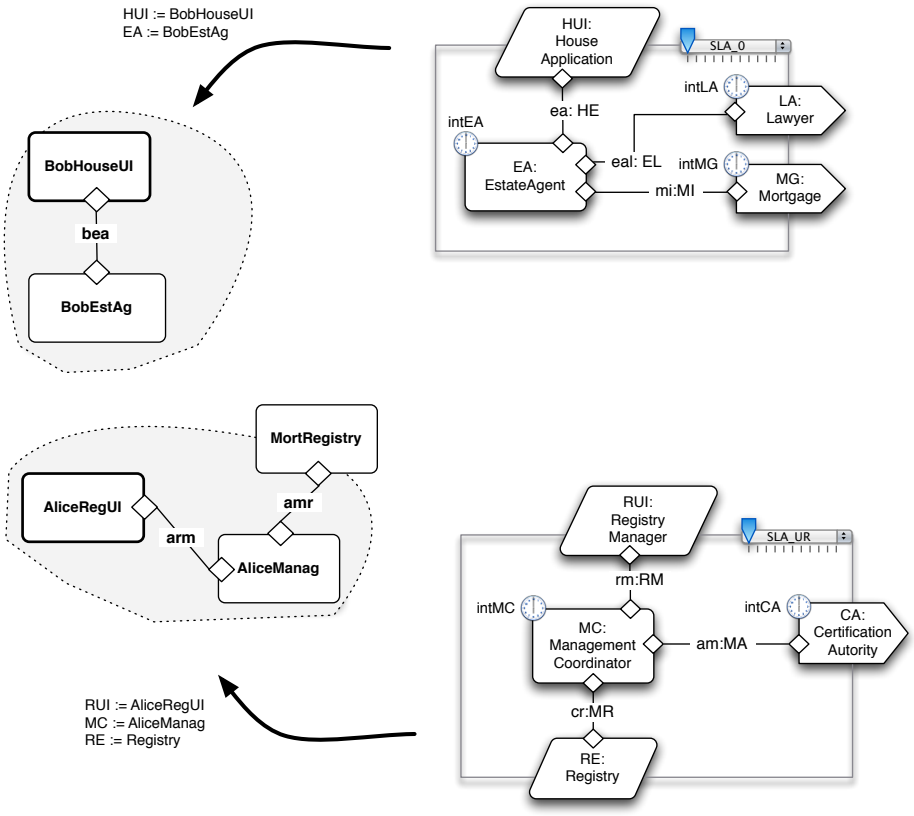

Fig. 5. A business conguration that precedes that of Figure 3.

notice that the new activity module that types A_{Bob} acquires the requires-interfaces of $MortgageAgent$, i.e. the business activity evolves both at the level of its configuration and its type.

5 Related Work

In the last decade, different approaches to architectural specification have been proposed that permit the representation of dynamic architectures [3,5,12,29,32,33]. The focus of these approaches is on the description of a control (reconfiguration) layer on top of a managed system. The dynamic architectural changes that have to be performed in the managed system are specified explicitly, for instance in terms of reconfiguration rules [5,12,32], configurator processes [3] or reconfiguration scripts [29,33]. Although different semantic domains have been used in those aforementioned works, their underlying mechanisms can be defined in terms of operations that rewrite state configurations in the sense of Definition 1. The work that we presented in this paper follows on this tradition but offers a more structured approach (based on reflection) that targets the forms of reconfiguration that arise, specifically, in SOC.

A different direction was taken by Darwin [24], π-ADL [27] and ARCHWARE [26], which explore the expressive power of the π-calculus — a calculus developed precisely for concurrent systems whose configurations may change during computation. As a result, these ADLs do not promote the separation between the management of the computational aspects of systems and of their architecture (configuration); by borrowing primitives from the π-calculus, they include instantiation, binding and rebinding as part of the behaviour of system components. From our point of view, the separation that the approaches mentioned in the previous paragraph (including ours) promote between the two levels (computation and reconfiguration) has clear advantages for managing the complexity that arises in modern software-intensive systems, especially when, like in SOC, their architecture is highly dynamic. The expressive power of the π-calculus has also been explored within SOC: several service calculi have been proposed to address operational foundations of SOC (in the sense of how services compute) [13,15,22,23] as well as to capture the dynamic architectures of service-oriented systems [28,31]. Here again, a clear separation between the aspects that belong to the SOA middleware and those that derive from the application domain seems to be essential for the definition of ADLs that can effectively support high-level design.

Therefore, the reason that led us to propose a different model for dynamic architectures specifically targeted for SOC is not the lack expressiveness of existing models but, rather, the lack of models that capture the 'business' aspects of SOC at the 'right' level of abstraction. To our knowledge, ours is the first proposal in this direction.

Indeed, the definition of models is intrinsically associated with *abstraction*. For example, operational models of sequential programming are typically defined in terms of functions (called states) that assign values to variables, which abstract from the way memory is organised and accessed in any concrete conventional computer architecture. Paradigms such as SOC superpose further layers of abstraction (creating a richer middleware) so that systems can be built and interconnected by relying on a software infrastructure that adds to the basic computation and communication platform a num-

ber of facilities that, in the case of SOAs, support service publication, discovery and binding. This means that designers or programmers working over a SOA do not need to implement these mechanisms: they can rely on the fact that they are available as part of the *abstract operating system* that is offered by the middleware. Just like any Java programmer does not need to program the dynamic allocation, referencing and dereferencing of names, a programmer of a complex service should not need to include the discovery, selection and binding processes among the tasks of the orchestrator.

This is why we perceive that the architectural aspects of SOC are best handled over graph-based representations that separate computation from reconfiguration such as the ones proposed in this paper. Drawing an analogy with the semantics of programming languages, we could say that we proposed a notion of (typed) state and state transition for such dynamic aspects of SOC: states are graphs of components and connectors that capture configurations that execute business activities, and transitions are reconfigurations that result from binding to selected services. Our model captures the nature of SOA-middleware approaches and generalises them, offering a more abstract level of modelling in which the business aspects that drive reconfiguration can be represented explicitly and separately from the orchestration of the interactions through which services are delivered.

6 Concluding Remarks

In this paper we presented a mathematical model that can be used as a semantic domain for service-oriented architectural description languages. The static aspects of our model were inspired by the concepts proposed in the Service Component Architecture (SCA) [25] towards a general assembly model and binding mechanisms for service components and clients that may have been programmed in possibly many different languages, e.g. Java, C++, BPEL, or PHP. We have transposed those concepts to a more abstract level of modelling and enriched them with primitives that address the dynamic aspects (run-time service discovery, selection and binding) of service-oriented systems. This model paves the way for the definition of ADLs that are able to address the specification of dynamic architectural characteristics of service-oriented applications and, moreover, contribute to overcome the lack of models that capture the 'business' aspects of SOC.

The advantages of this approach have been explored in the language SRML that we defined in SENSORIA [18] but our model is general enough that it can be used to support other ADLs. For example, at a methodological level, we have extended the traditional use-case method to define the structure of both activity and service modules from business requirements [9], which was validated in a number of case studies, including automotive [10] and telco systems [1] in addition to more classical business-oriented domains such as the one used in the paper. Another advantage of the separation of reconfiguration from computation is that different orchestration languages can be used for modelling the components and connectors through which services are provided without affecting the way activities or services are structured in modules: for example, transformations were defined from BPEL to SRML [11], UML state machines were used

for supporting model-checking [2], and transformations to PEPA [20] were used for supporting quantitative analysis [8].

Acknowledgments

We would like to thank our colleagues in the SENSORIA project for many useful discussions on the topics covered in this paper, in particular João Abreu and Laura Bocchi for their contribution to the definition of SRML.

References

1. J. Abreu, L. Bocchi, J. Fiadeiro, and A. Lopes. Specifying and Composing Interaction Protocols for Service-Oriented System Modelling. In *Formal Techniques for Networked and Distributed Systems*, volume 4574 of *LNCS*, pages 358–373. Springer, 2007.
2. J. Abreu, F. Mazzanti, J. Fiadeiro, and S. Gnesi. A Model-Checking Approach for Service Component Architectures. In *Formal Techniques for Distributed Systems*, volume 5522 of *LNCS*, pages 219–224. Springer, 2009.
3. R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *FASE*, pages 21–37, 1998.
4. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.
5. T. Batista, A. Joolia, and G. Coulson. Managing dynamic reconfiguration in component-based systems. In *EWSA*, pages 1–17, 2005.
6. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal ACM*, 44(2):201–236, 1997.
7. S. Bistarelli and F. Santini. A nonmonotonic soft concurrent constraint language for sla negotiation. *ENTCS*, 236:147–162, 2009.
8. L. Bocchi, J. Fiadeiro, S. Gilmore, J. Abreu, M. Solanki, and V. Vankayala. A formal approach to modelling time properties of service oriented systems. Submitted., 2009.
9. L. Bocchi, J. Fiadeiro, and A. Lopes. A Use-Case Driven Approach to Formal Service-Oriented Modelling. In *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *CCIS*, pages 155–169. Springer, 2008.
10. L. Bocchi, J. Fiadeiro, and A. Lopes. Service-oriented modelling of automotive systems. In *The 32nd Annual IEEE International on Computer Software and Applications, COMPSAC'08*, pages 1059–1064. IEEE, 2008.
11. L. Bocchi, Y. Hong, A. Lopes, and J. Fiadeiro. From bpel to srml: a formal transformational approach. In *Web Services and Formal Methods*, volume 4937 of *LNCS*, pages 92–107. Springer, 2008.
12. R. Bruni, A. Bucchiarone, S. Gnesi, D. Hirsch, and A. Lluch Lafuente. Graph-based design and analysis of dynamic software architectures. In *Concurrency, Graphs and Models*, volume 6065 of *LNCS*. Springer, 2008.
13. M. Carbone, K. Honda, and N. Yoshida. A calculus of global interaction based on session types. *ENTCS*, 171(3):127–151, 2007.
14. G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1):1–42, 2008.
15. M. Boreale et al. Scc: A service centered calculus. In *Web Services and Formal Methods*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006.

16. H. Fargier, J. Lang, R. Martin-Clouaire, and T. Schiex. A constraint satisfaction framework for decision under uncertainty. In *Proc. of the 11th Int. Conf. on Uncertainty in Artificial Intelligence*, pages 175–180, 1996.
17. J. Fiadeiro, A. Lopes, and L. Bocchi. An abstract model of service discovery and binding. Available from www.cs.le.ac.uk/people/jfiadeiro.
18. J. Fiadeiro, A. Lopes, L. Bocchi, and J. Abreu. The Sensoria reference modelling language. Available from www.cs.le.ac.uk/people/jfiadeiro.
19. D. Garlan, S-W. Cheng, A-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.
20. S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in LNCS, pages 353–368. Springer, 1994.
21. F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications ACM*, 45(6):33–38, 2002.
22. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *Programming Languages and Systems*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.
23. R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. In *Journal of Logic and Algebraic Programming*. Elsevier press, 2005.
24. J. Magee and J. Kramer. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, 1996.
25. Michael Beisiegel et al. Service Component Architecture Specifications, 2007.
26. R. Morrison, G. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cmpan, B. Warboys, B. Snowdon, and R. Greenwood. Support for evolving software architectures in the ArchWare ADL. In *4th Working IEEE/IFIP Conference on Software Architecture*, 2004.
27. F. Oquendo. π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes*, 29(3):1–14, 2004.
28. F. Oquendo. Formal approach for the development of business processes in terms of service-oriented architectures using pi-adl. In *SOSE*, pages 154–159, 2008.
29. P. Oreizy and R.Taylor. On the role of software architectures in runtime system reconfiguration. *IEEE Proceedings- Software Engineering*, 145(5):137–145, 1998.
30. D. Perry and L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.
31. M. López Sanz, Z. Qayyum, C. Cuesta, E. Marcos, and F. Oquendo. Representing service-oriented architectural models using pi-adl. In *ECSA*, pages 273–280, 2008.
32. M. Wermelinger and J. Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Sci. Comput. Program.*, 44(2):133–155, 2002.
33. M. Wermelinger, A. Lopes, and J. Fiadeiro. A graph based architectural (re)configuration language. In *ESEC/FSE-9*, pages 21–32, New York, NY, USA, 2001. ACM.