Softw Syst Model
DOI 10.1007/s10270-012-0236-1

SPECIAL SECTION PAPER

A model for dynamic reconfiguration in service-oriented architectures

José Luiz Fiadeiro · Antónia Lopes

Received: 22 November 2010 / Revised: 17 November 2011 / Accepted: 6 February 2012

Abstract The importance of modelling the dynamic characteristics of the architecture of software systems has long been recognised. However, the nature of the dynamics of service-oriented applications goes beyond what is currently addressed by architecture description languages (ADLs). At the heart of the service-oriented approach is the logical separation between the service need and the need-fulfillment mechanism, i.e., the provision of the service: the binding between the requester and the provider is deferred to run time and established at the instance level, i.e., each time the need for the service arises. As a consequence, computation in the context of service-oriented architectures transforms not only the states of the components that implement applications but also the configurations of those applications. In this paper, we present a model for dynamic reconfiguration that is general enough to support the definition of ADLs that are able to address the full dynamics of service-oriented applications. As an instance of the model, we present a simple service-oriented ADL derived from the modelling language SRMLthat we developed in the SENSORIA project.

Keywords Software architecture · Service-oriented computing · Dynamic formal modelling

Communicated by Dr. Muhammad Ali Babar, Flavio Oquendo, and Ian Gorton.

J. L. Fiadeiro
Department of Computer Science,
University of Leicester, University Road, Leicester LE1 7RH, UK
e-mail: jose@mcs.le.ac.uk

A. Lopes (✉)
Faculty of Sciences, University of Lisbon, Campo Grande,
1749-016 Lisbon, Portugal
e-mail: mal@di.fc.ul.pt

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 the 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" [38], 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 also 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 considerable attention from the research community [3,26,33,34,36,43], mainly as a response to the need for mechanisms that can enhance 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 the context of SOC is needed.

Indeed, dynamic reconfiguration is 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 architecture-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.

In this paper, we address the lack of architectural models that can cope with the intrinsic dynamicity of service configuration by showing how the mathematical model that we proposed in [24] for service discovery and binding can be used as a semantic domain for service-oriented architectural description languages. Mathematical models that capture the essence of a paradigm play an essential role as a foundation for methods, languages and support tools for that paradigm [13]. Architectural models in particular contribute to the identification of abstractions that are useful for describing the architecture of software systems, including the architecture of specific classes of systems. It has now become consensual that a general purpose ADL must include explicit support for modelling both component and connector types as well as configurations, for example as graphs whose nodes are typed by component and connector types and whose edges represent attachments. In order to understand how SOC impacts on those configurations, and how this differs from the other forms of dynamic reconfiguration mentioned above, we need to resort to a mathematical model that offers a layer of abstraction at which we can capture the nature and analyse the properties of the transformations that, under SOC, operate on configurations. An example of how an operational account of dynamic reconfiguration under service-oriented architectures can be defined over our mathematical model can be found in [15].

The mathematical model proposed in this paper was used in the SENSORIA project to define the dynamic semantics of the language SRML [25] and support quantitative (e.g., [7]) and qualitative (e.g., [2]) analysis techniques. Herein, we define a 'light' version of SRMLthat is simpler but expressive enough to be used as an ADL for service-oriented applications. In this language, we use a linear-time temporal logic (LTL) for specifying the behaviour of the components and interaction protocols through which business systems are configured and services are delivered. In what concerns the dynamic aspects, we adopt a graph-based approach in the tradition of what we and other authors have used for architectural reconfiguration [14,43]. Essentially, we introduce a mechanism of reflection (as used in other approaches to dynamic reconfiguration [17,28,30,41,42]) by which configurations are typed with models of business activities and service models define rules for dynamic reconfiguration.

More precisely, the interface between the structural and dynamic aspects of our approach operates as follows: (1) LTL specifications are used for describing the behaviour of components and interactions between them, and also for specifying the behaviour required of, or offered by, services (what in Sect. 5 we call interfaces); (2) structures (which we call modules) of such specifications are used for typing the configurations of systems—they describe the behaviour of the components and interconnections present in the configuration and they identify the interfaces to external services that may need to be discovered; (3) modules also encapsulate rules for evolving the configuration when the discovery of a required service is triggered. At the same time that the configuration is changed, a new module is calculated that is used for typing the resulting configuration (which is what makes our model reflective).

The paper is organised as follows. In Sect. 2, we provide the basic concepts that underly our view of dynamic reconfiguration in service-oriented architectures and introduce the example that we use throughout the paper for illustrating our approach (a financial case study developed in SENSORIA). In Sect. 3, we introduce `SRMLight` and its role in this paper. In Sect. 4, we define a model for state configurations of systems and, in Sect. 5, we define a model for business-reflective configurations of systems. In Sect. 6, 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 Sect. 7 and conclude in Sect. 8 by pointing to other aspects of SOC that we have been investigating.

2 Preliminaries

2.1 Configurations of global computers

As mentioned in Sect. 1, SOC defies the traditional notion of software system. Whereas, in the context of component-based development, we can design a system "by assembling prefabricated software components" [19], within SOC it is not possible to define, at design time, what the architecture of a system will be because it will evolve, at run time, as applications trigger the discovery of services and bind to them to fulfil given business goals. Therefore, one needs to think that computation takes place in the context of 'global computers'—computational infrastructures that are available globally and support the distributed execution of business applications—and that computation in this context transforms not only the states of the components that implement applications but also the configurations of those applications.

At a certain level of abstraction, the configuration of such a global computer on a given moment of time 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]

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

We denote by **COMP** and **WIRE** the universes of components and wires, respectively.

Definition 1 (*Configuration*) A *configuration* is a simple graph $\mathcal{G}$ such that $nodes(\mathcal{G}) \subseteq \mathbf{COMP}$ (i.e., nodes are components) and $edges(\mathcal{G}) \subseteq \mathbf{WIRE}$ (i.e., edges are wires). Each edge w is a (unordered) pair of nodes that we denote by $c_1 \overset{w}{\leftrightarrow} c_2$.

The fact that the graph is simple—undirected, without self-loops or multiple edges—means that all interactions between two components are supported by a single wire and that no component can interact with itself (components are instances; different components of the same type—i.e., that implement the same specification—can interact with each other). The graph is undirected because typical service-oriented interactions are conversational, i.e., the wires need to be able to transmit messages both ways. These configurations are global in the sense that they apply to a given global computer. In Sect. 5, we refine this notion by recognising sub-configurations that are local to given business applications.

2.2 Running example

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.

As a running example, we use a (simplified) scenario in which there is a financial services organisation that offers a mortgage-brokerage service (called 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* and 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 Fig. 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

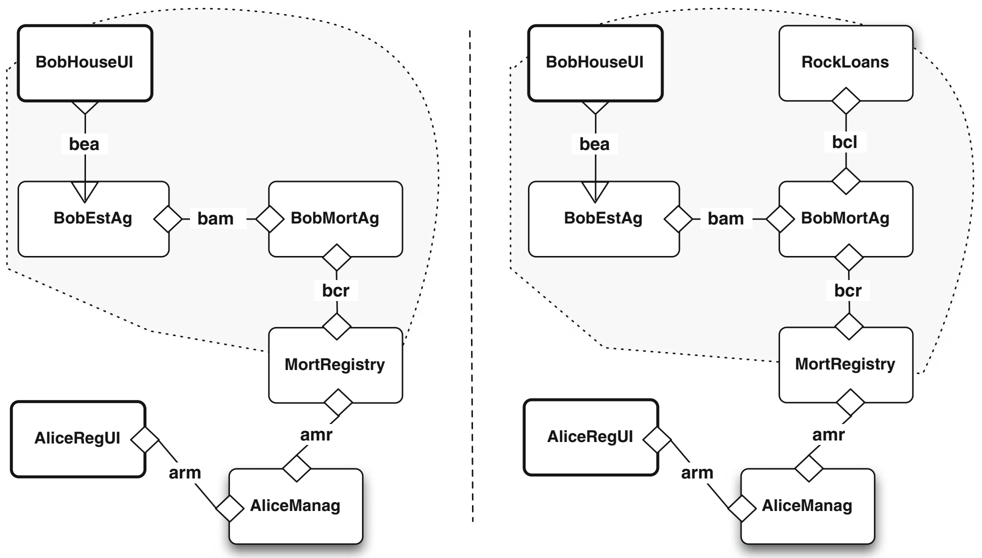

Fig. 1 Two configurations that shows the sub-configuration that corresponds to the business activity A_{Bob} before and after the discovery of a provider of the service *Lender*, respectively

using the interface *BobHouseUI*), and *MortRegistry*, a database 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 Fig. 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 Fig. 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 orchestrating the delivery of MORTGAGEFINDER. This new component is responsible for the provision of the service by the selected provider of *Lender*.

This illustrates why, in order to capture the dynamic aspects of SOC, we need to look beyond the information available in configurations. They 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.

3 SRMLight

Our goal in this paper is to offer a formal model for the reconfiguration steps that arise in service-oriented systems. This model is defined in terms of abstractions like **COMP** and **WIRE** so that it is independent of the nature of components and wires and of the underlying computation and communication model. In order to illustrate how this model can be used to support the definition of ADLs for service-oriented systems, we use SRMLight. This language, though not a full-fledged ADL, is rich enough to illustrate the use of the model and to provide a better insight on its capabilities.

SRMLight is a 'light' version of the modelling language SRML [25] that was developed as part of the SENSORIA project [45] within which it was validated over a number of case studies (e.g., in telecommunications [1], automotive [9], and financial systems [25]). The simplification retains all the essential elements of the execution and reconfiguration models of SRML, which can be found in [23]. A detailed account of the mortgage brokerage service description in SRML can be found in [25].

In SRMLight, components can either interact asynchronously by exchanging messages, or synchronously by reading from or writing into their states. Asynchronous communication is essential for supporting the forms of loose binding that are typical of SOC, whereas synchronous interactions are useful for tighter binding with persistent components such as databases or software-enabled devices (sensors, GPSs, and so on).

In order to handle data, we assume a fixed data space that we model as a pair $\Sigma = \langle D, F \rangle$ where D is a set of data sorts (such as int, $currency$, and so on) and F is a $D^* \times D$-indexed family of sets of operations over the sorts. We also assume a fixed partial algebra $\mathcal{U}$ for interpreting Σ [39]. Partiality arises from the fact that variables or parameters may become defined only when certain events occur. We use $\perp$ to represent the undefined value and work with strong equality, i.e., two terms need to be defined in order to be equal.

Every architectural element has a signature, which defines the messages and variables that characterise their behaviour. Messages may have a number of parameters in order to transmit data.

Every message m has a finite set P_m of *parameters* each of which is a pair $\langle p, d\rangle$ where p is the name of the parameter and $d \in D$ is the type of p.

A *component signature* is a pair $\langle \mathsf{V}, \mathsf{M}\rangle$, where:

– V is a finite set of variables each of which is a pair $\langle v, d\rangle$ where v is the name of the variable and $d \in D$ is the type of v.
– M is a finite set of messages partitioned in two sets M^- and M^+. The messages in M^- are said to have polarity $^-$ (meaning that they are outgoing), and those in M^+ have polarity $^+$ (meaning that they are incoming).

For example, the signature of the component *BobMortAg* of the running example might include messages that account for requests for a proposal (*reqP*) from a customer, replying to the customer (*replyP*), asking a lender for a quote (*askQ*) or receiving a quote from a lender (*recQ*).

V – *S:[init,waiting,replied], seqNum:nat, lender:set_of(lender_id), charge:nat*
M^+ – *reqP*, *recQ*

P_{reqP} – *usr:usr_data*, *income:money*
P_{recQ} – *prop:mortgage, loan:loan_data, accountRequired:bool*

M^- – *askQ*, *replyP*

P_{askQ} – *usr:usr_data*, *income:money, id:nat*
P_{replyP} – *prop:mortgage, cost:money*

An *A-wire signature* is a finite set M of messages.

Notice that the messages of A-wires do not have a polarity because the role of A-wires is to transmit messages between components and, hence, each message is both incoming and outgoing. In addition to a signature, every A-wire $c_1 \overset{w}{\leftrightarrow} c_2$ has an associated *connection*, which interconnects the components c_1 and c_2.

Let $c_1 \overset{w}{\leftrightarrow} c_2$ be an A-wire with signature M, and $\langle \mathsf{V}_i, \mathsf{M}_i\rangle$ the signature of c_i $(i = 1, 2)$. The connection associated with w is a pair of injections $\mu_i : \mathsf{M} \rightarrow \mathsf{M}_i$ such that $\mu_i^{-1}(\mathsf{M}_i^+) = \mu_j^{-1}(\mathsf{M}_j^-)$, $\{i, j\} = \{1, 2\}$. Each injection μ_i is called the *attachment* of w to c_i. We denote the connection by $(c_1 \overset{\mu_1}{\leftarrow} w \overset{\mu_2}{\rightarrow} c_2)$.

The condition $\mu_i^{-1}(\mathsf{M}_i^+) = \mu_j^{-1}(\mathsf{M}_j^-)$ means that the wire connects messages that have opposite polarities, i.e., an incoming message of one component is connected by the wire to an outgoing message of the other component.

Notice that the injections are not identities: the components and the wire may all use different names for the same message. This is important because, in the context of SOC, it is not possible to anticipate which names will be used by services discovered and bound to at run time. Therefore, interconnections are established explicitly by the attachments, not implicitly by name sharing.

As an example, suppose that *RockLoans* is a component whose signature includes messages that account for requests for a mortgage quote (*reqM*) and replies to those requests (*replyM*):

M^+ – *reqM*

P_{reqM} – *usr:usr_data*, *income:money*, *id:nat*

M^- – *replyM*

P_{replyM} – *prop:mortgage, loan:loan_data, accountRequired:bool*

The signature of the A-wire *bcl* used for connecting *BobMortAg* and *RockLoans* might then include:

M – *req*, *reply*

P_{req} – *usr:usr_data*, *income:money*, *id:nat*
P_{reply} – *prop:mortgage, loan:loan_data, accountRequired:bool*

The connection associated with the wire *bcl* would then involve, as attachments, the injections

$$\mu_1 : req \mapsto askQ, reply \mapsto recQ$$
$$\mu_2 : req \mapsto reqM, reply \mapsto replyM$$

Notice that, as required, $askQ$ and $reqM$, which are connected via req, do have opposite polarities: $askQ$ is outgoing for $BobMortAg$ and incoming for $RockLoans$ (ditto for $replyM$ and $recQ$).

A-wires do not connect variables, just messages. On the other hand, S-wires connect only variables. An *S-wire signature* is a finite set V of variables. Let $c_1 \overset{w}{\leftrightarrow} c_2$ be an S-wire with signature V, and $\langle \mathsf{V}_i, \mathsf{M}_i\rangle$ the signature of c_i $(i = 1, 2)$. The connection associated with w is a pair of partial injections $\mu_i : \mathsf{V} \leftarrow \mathsf{V}_i$ such that $\mu_1(\mathsf{V}_1) \cap \mu_2(\mathsf{V}_2) = \emptyset$ and $\mu_1(\mathsf{V}_1) \cup \mu_2(\mathsf{V}_2) = \mathsf{V}$. We denote this connection by $(c_1 \overset{\mu_1}{\rightarrow} w \overset{\mu_2}{\leftarrow} c_2)$.

As an example, assume that the signature of the component *MortRegistry*, a database, includes:

V – *selectedLenders:set_of(lender_id), reqNum:nat*

The signature of the S-wire *bcr* that connects *BobMortAg* and *MortRegistry* would then include

V – *le1,le2:set_of(lender_id), seq1,seq2:nat*,

so that the connection is established via the following partial injections:

μ_1 : *lender* $\mapsto$ *le1*, *seqNum* $\mapsto$ *seq1*
μ_2 : *selectedLender* $\mapsto$ *le2*, *reqNum* $\mapsto$ *seq2*

Note that partiality is essential on the side of *BobMortAg*, whose signature has more variables than those being connected to *MortRegistry*.

4 State configurations

In Sect. 2, we defined configurations of global computers as graphs of components linked through wires. In order to account for the way configurations evolve, it is necessary to consider the states of the configuration elements and the steps that they can execute.

For this purpose, we take that every component $c \in \mathbf{COMP}$ and wire $w \in \mathbf{WIRE}$ of a configuration may be in a number of states, the set of which is denoted by $\mathbf{STATE}_c$ and $\mathbf{STATE}_w$, respectively. We further assume that, for every component $c \in \mathbf{COMP}$ and wire $w \in \mathbf{WIRE}$, there are subsets $\mathbf{STATE}^{\,0}_{c}$ and $\mathbf{STATE}^{\,0}_{w}$ of initial states.

In SRMLight, states consist of the values taken by a number of typed variables and buffers (modelled as sets) where they store the messages that are pending to be processed. More precisely, a *state* of a component c with signature $\langle \mathsf{V}, \mathsf{M}\rangle$ (an element of $\mathbf{STATE}_c$) is a pair $\langle \mathcal{VAL}, \mathcal{INV}\rangle$, where:

– $\mathcal{VAL}$ assigns to every $\langle v, d\rangle \in \mathsf{V}$ and $\langle p, d\rangle \in \mathsf{P}_m$, $m \in \mathsf{M}$, an element of $d_{\mathcal{U}}$.
– $\mathcal{INV} \subseteq \mathsf{M}^+$ is the set of messages that have been invoked and are waiting to be processed.

We use sets as buffers because components execute in 'sessions' and, within a session, messages are not repeated.

For example, in *BobMortAg* we use the variable *S* to model a state machine that captures the workflow executed by the component. A possible initial state might assign *init* to *S* and undefined values ($\perp$) to the other variables and message parameters.

A *state* for an A-wire w with signature M (an element of $\mathbf{STATE}_w$) is a pair $\langle \mathcal{VAL}, \mathcal{PND}\rangle$ where $\mathcal{PND} \subseteq \mathsf{M}$—the set of messages that are pending to be delivered by the wire—and $\mathcal{VAL}$ assigns values to the parameters of the messages as in the case of states for component signatures. On the other hand, because they are synchronous, S-wires are stateless.

Based on state abstractions for components and wires—$\mathbf{STATE}_c$ and $\mathbf{STATE}_w$—we can define a notion of state for a whole configuration:

Definition 2 (*State configuration*) A *state configuration* 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 $w \in edges(\mathcal{G})$.

A state configuration $\langle \mathcal{G}, \mathcal{S}\rangle$ may change in two different ways:

– A state transition from $\mathcal{S}$ to $\mathcal{S}'$ can take place within the configuration $\mathcal{G}$; we call such transitions *execution steps*. An execution step involves a local transition at the level of each component and wire, though some may be idle.
– Both a state transition from $\mathcal{S}$ to $\mathcal{S}'$ and a change from the configuration $\mathcal{G}$ to another configuration $\mathcal{G}'$ can take place; we call such transitions *reconfiguration steps*.

In this section, we discuss execution steps, leaving reconfiguration steps to Sect. 6.

The nature of execution steps is abstracted away by assuming that, for every $c \in \mathbf{COMP}$ (resp. $w \in \mathbf{WIRE}$), and every pair of states $s, s' \in \mathbf{STATE}_c$ (resp. $\mathbf{STATE}_w$), there is a family $\mathbf{STEP}_c^{s\to s'}$ (resp. $\mathbf{STEP}_w^{s\to s'}$) of steps from s to s'.

In the particular case of SRMLight, an execution step in $\mathbf{STEP}_c^{\langle \mathcal{VAL},\mathcal{INV}\rangle\to\langle \mathcal{VAL}',\mathcal{INV}'\rangle}$ for a component c with signature $\langle \mathsf{V}, \mathsf{M}\rangle$ consists of four sets of messages $\mathcal{PRC}$, $\mathcal{EXC}$, $\mathcal{DLV}$ and $\mathcal{PUB}$ such that:

– $\mathcal{PRC} \subseteq \mathcal{INV}$ is the set of messages that are selected to be processed during the step.
– $\mathcal{EXC} \subseteq \mathcal{PRC}$ is the set of selected messages that are actually executed (the remaining ones are discarded).
– $\mathcal{DLV} \subseteq \mathsf{M}^+$ is the set of messages that are delivered to the component during the step.
– $\mathcal{PUB} \subseteq \mathsf{M}^-$ is the set of messages that are published by the component during the step.
– $\mathcal{INV}' = (\mathcal{INV} \setminus \mathcal{PRC}) \cup \mathcal{DLV}$

That is, the state is changed by removing from the buffer the messages selected to be processed (executed or discarded) and adding those that are delivered to the component. The variable valuation can also change, of course.

An execution step in $\mathbf{STEP}_w^{\langle \mathcal{VAL},\mathcal{INV}\rangle\to\langle \mathcal{VAL}',\mathcal{INV}'\rangle}$ for an A-wire w with signature M consists of two sets of messages $\mathcal{DLV}$ and $\mathcal{PUB}$ such that:

– $\mathcal{DLV} \subseteq \mathcal{PND}$ is the set of messages that are delivered by the wire during the step.
– $\mathcal{PUB}$ is the set of messages that are published to the wire during the step.
– $\mathcal{PND}' = (\mathcal{PND} \setminus \mathcal{DLV}) \cup \mathcal{PUB}$

That is, the state is changed by removing from the buffer the messages that are delivered to the components and adding those that are published to the wire.

It remains to define how components and wires perform joint execution steps as part of a configuration.

Definition 3 (*Configuration execution step*) Given a configuration $\mathcal{G}$, an execution step between a pair of states $\mathcal{S}$ and $\mathcal{S}'$ consists of a mapping $\mathcal{T}$ that assigns to every component c (resp. wire w), a step in $\mathbf{STEP}_c^{\mathcal{S}_c \to \mathcal{S}'_c}$ (resp. $\mathbf{STEP}_w^{\mathcal{S}_w \to \mathcal{S}'_w}$) subject to a number of constraints that ensure that the execution step of every wire agrees with the execution steps of the components that the wire connects.

The specific contraints that, SRMLight, apply to execution steps are, for every connection $(c_1 \overset{\mu_1}{\leftarrow} w \overset{\mu_2}{\rightarrow} c_2)$:

– $\mathcal{PUB}_w = \mu_1^{-1}(\mathcal{PUB}_{c_1}) \cup \mu_2^{-1}(\mathcal{PUB}_{c_2})$
– $\mathcal{DLV}_w = \mu_1^{-1}(\mathcal{DLV}_{c_1}) \cup \mu_2^{-1}(\mathcal{DLV}_{c_2})$

That is, every wire delivers all messages to (and only to) the components it connects, and all the messages that are published to the wire come from the same components.

Definition 4 (*Execution path and behaviour*) An execution path for a component or wire is an infinite sequence of alternating states and steps $\lambda = \langle s_0 t_0 s_1 t_1 \cdots \rangle$ such that s_0 is an initial state and, for every i, t_i is an execution step between s_i and s_{i+1}. The behaviour of a component c (resp. wire w) is a set Λ_c (resp. Λ_w) of execution paths.

Given an execution path $\lambda = \langle s_0 t_0 s_1 t_1 \cdots \rangle$, we denote by λ^i the ith suffix of the path λ, i.e. the infinite sequence that starts in state s_i.

Definition 5 (*Configuration execution path and behaviour*) An execution path for a configuration $\mathcal{G}$ consists of an infinite sequence $\lambda = \langle \mathcal{S}_1 \mathcal{T}_1 \mathcal{S}_2 \mathcal{T}_2 \cdots \rangle$ of alternating states and steps and, for every component c (resp. wire w), an index i_c (resp. i_w). The behaviour of a configuration $\mathcal{G}$, which we denote by $\Lambda_{\mathcal{G}}$, consists of all the execution paths λ such that the projection of λ^{i_c} (resp. λ^{i_w}) into the states and steps of c (resp. w), belongs to Λ_c (resp. Λ_w).

The reason for the indexes is that we need to take into account the fact that, as exemplified in Sect. 6, configurations may evolve through the addition of components or wires. Therefore, we need to account for the state in which components and wires join a configuration.

5 Business-reflective configurations

In order to capture the business activities that perform in a configuration and determine how the configuration evolves, we need a more sophisticated typing mechanism that goes beyond the typing of the individual components and wires: we need to capture activities as more complex structures of specifications, which we call *activity modules*—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 Fig. 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) through which the activity interacts 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 specification of the interaction protocols that are executed by the wires.
– Requires-interfaces: the specifications of the external services that may be required during the execution of the activity. For instance, the activity module in Fig. 2 declares three requires-interfaces—*LA* of type *Lawyer*, *IN* of type *Insurance*, *LE* of type *Lender* and *BA* of type

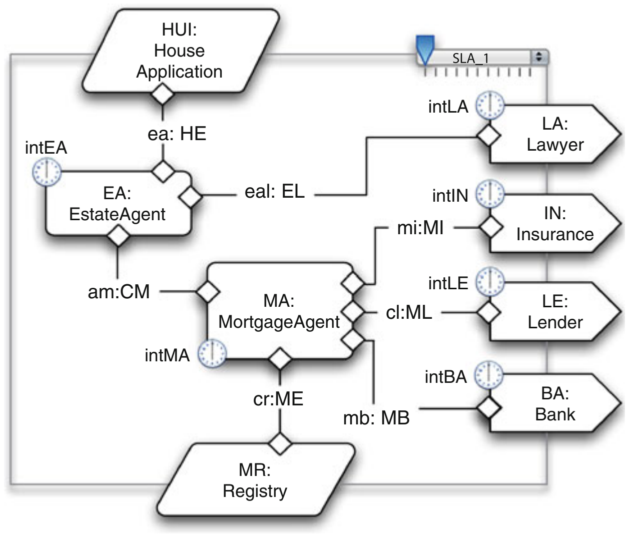

Fig. 2 The activity module that types the sub-configuration that corresponds to A_{Bob} as shown on the *left-hand side* of Fig. 1

Bank. These types 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 clocks.
– External configuration policies: these are constraints through which service-level agreements (SLAs) can be negotiated with external services during discovery and selection. Graphically, these policies are identified by rulers.

The overall framework that we propose is largely independent of the choice of a specification language and formalism—we distinguish between the different kinds of interfaces because they have different roles in the dynamic re-configuration of the activity as explained in Sect. 6. Therefore, for generality, we may assume that all specifications belong to a universe **SPEC** and that we have available a satisfaction relation $\models$ between execution paths (in the sense of Definition 4) and specifications.

A component c (resp. wire w) implements a specification Φ if, for every execution path λ of Λ_c (resp. Λ_w), $\lambda \models \Phi$.

In SRMLight we use a single formalism for specifying components and wires—a linear-time version of UCTL, a logic that we used in [2,23] for giving semantics to, and model checking, SRML specifications. We opted for a linear-time logic in SRMLight because it is simpler than UCTL and can build directly on the notion of execution path introduced in Definition 4.

The formal definition of this logic is presented in Appendix A. In this logic, both states and transitions are labelled and formulas $m!, m¡, m?, m¿$ refer to the action of publishing, delivering, executing and discarding a message m, respectively. In the next paragraphs, we provide examples of specifications in SRMLightfor some of the components and wires involved in the activity module presented in Fig. 2.

For example, the specification *MortgageAgent* of *BobMortAg* might include the following properties:

$$
\begin{aligned}
&\Box(S = init \wedge reqP¡ \supset \Diamond reqP?)\\
&\Box([reqP?](S = waiting\\
&\qquad \wedge askQ!\\
&\qquad \wedge askQ.usr = reqP.usr\\
&\qquad \wedge askQ.income = reqP.income\\
&\qquad \wedge askQ.id = seqNum))\\
&\Box(S = waiting \wedge recQ¡ \supset \Diamond recQ?)\\
&\Box([recQ?](S = replied\\
&\qquad \wedge replyP!\\
&\qquad \wedge replyP.prop = recQ.prop\\
&\qquad \wedge replyP.cost = (1 + charge/100) * 750))
\end{aligned}
$$

The first two properties state that the component ensures to execute the request for a proposal (*reqP*) if the request is delivered when S is $init$, and that the execution publishes the message *askQ*, i.e., a request for a quote. The parameters *usr* and *income* of *askQ* are the same as those of the *reqP*, and the value of *id* is provided by the variable *seqNum*. Moreover, the execution of *reqP* changes S to $waiting$.

The last two properties state that the component ensures to act upon receiving the quote (*recQ*) if delivered when S is $waiting$, as a result of which it publishes *replyP* with the parameter *prop* set to the value of the corresponding parameter of *recQ*, and *cost* calculated in terms of the variable *charge*. This *charge* is applied to the base price of the brokerage service (750). Moreover, S is changed to $replied$.

As already mentioned, we say that *BobMortAg* implements *MortgageAgent* if all the execution paths of the component satisfy the specification, which is equivalent to

$$\Lambda_{BobMortAg} \subseteq \Lambda_{MortgageAgent}$$

In SRMLight, the specification *ML* of the wire *bcl* might include the following properties:

$$
\begin{aligned}
&\Box(BobMortAg.askQ! \supset \Diamond(RockLoans.reqM¡\\
&\qquad \wedge RockLoans.reqM.usr =\\
&\qquad\qquad BobMortAg.askQ.usr\\
&\qquad \wedge RockLoans.reqM.income =\\
&\qquad\qquad BobMortAg.askQ.income\\
&\qquad \wedge RockLoans.reqM.id =\\
&\qquad\qquad BobMortAg.askQ.id))\\
&\Box(RockLoans.replyM! \supset \Diamond(BobMortAg.recQ¡\\
&\qquad \wedge BobMortAg.recQ.prop =\\
&\qquad\qquad RockLoans.replyM.prop\\
&\qquad \wedge BobMortAg.recQ.loan =\\
&\qquad\qquad RockLoans.replyM.loan\\
&\qquad \wedge BobMortAg.recQ.accountRequired =\\
&\qquad\qquad RockLoans.replyM.accountRequired))
\end{aligned}
$$

These properties state that the wire will eventually deliver (i) the message *askQ*, sent by *BobMortAg*, to *RockLoans* under the local name *reqM*, and (ii) the message *replyM*, published by *RockLoans*, to *BobMortAg* under the name *recQ*. In this example, the parameters of the messages on each side coincide but, more generally, the connection may be required to operate conversions between the data transmitted through the variable (for example, currency or units of measurement).

A specification *ME* for the S-wire *bcr* might consist simply of the properties:

$$
\begin{aligned}
&\Box(BobMortAg.lender = MortRegistry.selectedLender)\\
&\Box(BobMortAg.seqNum = MortRegistry.reqNum)
\end{aligned}
$$

More complex examples could involve the operations of the data types to convert between data.

Notice that, in conjunction with the specification of *BobMortAg*, this connection implies that the value of the parameter *id* of the message *askQ* sent by *BobMortAg* to *RockLoans* through the A-wire specified above is stored in *MortRegistry*.

In the previous paragraphs, we have presented several specifications of components and wires interfaces used in the activity module of Fig. 2. Because requires-interfaces also play a critical role in activity modules, it is important to illustrate how they can be specified.

Consider, for instance, the requires-interface *Lender* of the activity module presented in Fig. 2. *Lender* refers to a specification with the same signature as *RockLoans* that includes, among other properties:

$$\Box(reqM¡ \supset \Diamond replyM!)$$

That is, the specification *Lender* requires the service to be discovered to reply, eventually, to any request for a mortgage quote.

The proposed framework is also independent of the language used for specifying initialisation conditions and triggers: we assume that we have available a set **STC** of conditions and a set **TRG** of triggers, and satisfaction relations between them and states and execution steps, respectively. State and trigger conditions in `SRMLight` are state formulas over the values of variables and action formulas involving the delivery of messages, respectively. Their formal definition is also presented in the appendix.

For example, in the activity module of Fig. 2, the initialisation condition for *MA:MortgageAgent* is simply

$$(state = init)$$

Furthermore, *LE:Lender* has to be discovered as soon as *reqP* is delivered to *MA:MortgageAgent* provided that MA is in the initial state. The trigger associated with the required interface *LE* is, therefore,

$$\langle (state = init), MA.reqP¡ \rangle$$

For SLA constraints, we adopt so called 'soft constraints', which 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 preferences, costs, inter alia. In particular, they have shown to be useful for supporting the negotiation of service-level agreements [6]. Some well-known soft constraint formalisms are *Valued Constraint Satisfaction Problems* [20] and *Semiring-based Soft Constraints* [5]. The particular formalism that is adopted is not relevant for the approach that we propose. For `SRMLight`, we use [5] in much the same way we used it in `SRML` [25].

Definition 6 (*c-semiring*) A c-semiring is a semiring of the form $\langle A, +, \times, 0, 1\rangle$ 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 — $a \leq_S b$ iff $a + b = b$. That is, b is better than a iff the choice between a and b is b.

Definition 7 (*Constraint system*) 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 V 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.

In `SRMLight`, because we want to handle constraints that involve different degrees of satisfaction, it makes sense that we work with the c-semiring $\langle [0..1], max, min, 0, 1\rangle$ of soft fuzzy constraints. In this c-semiring, the preference level is between 0 (worst) and 1 (best). `SRMLight` provides a set of standard configuration variables, namely:

- $m.UseBy$, for every outgoing message m; its value is the length of time the message is valid after it is issued.
- *ServiceId*, for every required-interface; it represents the identification of the service that is bound to that interface (for instance, a URI).

In addition, constraints can involve the variables declared in signatures. In the activity module presented in Fig. 2 we might include the following constraint involving the variable *MA.charge*:

$$C_1 : \{MA.charge,\ MA.replyP.UseBy\}$$
$$def(c, t) = \begin{cases} 1 & \textit{if } t = 10 \textit{ and } c = 1 \\ 0 & \textit{otherwise} \end{cases}$$

This constraint states that the charge applied to the base price of the brokerage service and the interval during which the proposal is valid are fixed to 10 and 1, respectively. This is, in fact, a constraint that has resulted from a negotiation performed during a reconfiguration step as discussed in the next section (in which we also illustrate the use soft constraints).

We might also include in the external policy of the activity module a constraint requiring the choice of the lender to be a member of *MA.lender*. This would be expressed as follows:

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

Recall that, according to the specification *MortgageAgent*, *MA.lender* is a set of trusted lenders obtained from the *Registry*.

Activity modules are formalised as graphs:

Definition 8 (*Activity module*) An *activity 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 $serves(M)$ not in $(requires(M) \cup uses(M))$. We use $components(M)$ to denote the set of all remaining nodes.
- A labelling function SP_M such that assigns a specification to every node and edge.
- A pair $intPlc(M)$ of mappings $\langle trigger_M, init_M\rangle$ such that $trigger_M$ assigns a condition in **TRG** to each node in $requires(M)$ and $init_M$ assigns a condition in **STC** to each node in $components(M)$.
- A pair $extPlc(M)$ of a constraint system $cs(M)$ and a set $sla(M)$ of 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 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 9 (*Business configuration*) A *business configuration* is a triple $\langle \mathcal{G}, \mathcal{B}, \mathcal{C}\rangle$ where

- $\mathcal{G}$ is a configuration (see Def. 1).
- $\mathcal{B}$ is a partial mapping that assigns an activity module $\mathcal{B}(a)$ to every activity $a \in \mathcal{A}$ — the workflow being executed by a. We say that the activities in the domain of this mapping are active.
- $\mathcal{C}$ is a mapping that assigns a homomorphism $\mathcal{C}(a)$ of graphs $body(\mathcal{B}(a)) \rightarrow \mathcal{G}$ to every activity $a \in \mathcal{A}$ that is active. We denote by $\mathcal{G}(a)$ the image of $\mathcal{C}(a)$—the sub-configuration of $\mathcal{G}$ that corresponds to the activity a.
- Every homomorphism $\mathcal{C}(a)$ is such that, for every node n (resp. edge e), $\mathcal{C}(a)(n)$ (resp. wire $\mathcal{C}(a)(e)$) implements $\mathsf{SP}_{\mathcal{B}(a)}(n)$ (resp. $\mathsf{SP}_{\mathcal{B}(a)}(e)$) and every initial state of $\mathcal{C}(a)(n)$ satisfies $init_{\mathcal{B}(a)}(n)$.
- Every node and edge of $\mathcal{G}$ belongs to at least an activity.

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) and the edges (wires) of $\mathcal{G}(a)$ with specifications of the roles that they play (implement) in the activity.

In Fig. 3, we represent a business configuration for the configuration depicted on the left-hand side of Fig. 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, Fig. 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 configuration 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 configuration. Indeed, as discussed above, the purpose of requires-interfaces is to identify the need that the activity may incur on external services if certain conditions become true (the triggers associated with the requires-interfaces). In particular, this makes requires-interfaces different from uses-interfaces as the latter are indeed mapped, through the homomorphism, to a component of the configuration.

In summary, the homomorphism makes configurations reflective in the sense of [17] as it adds meta (business) information to the 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.

6 Service binding as a reconfiguration action

Business configurations change whenever the execution of an activity requires the discovery of and binding to a service. It remains to formalise this process, which starts with the dis-

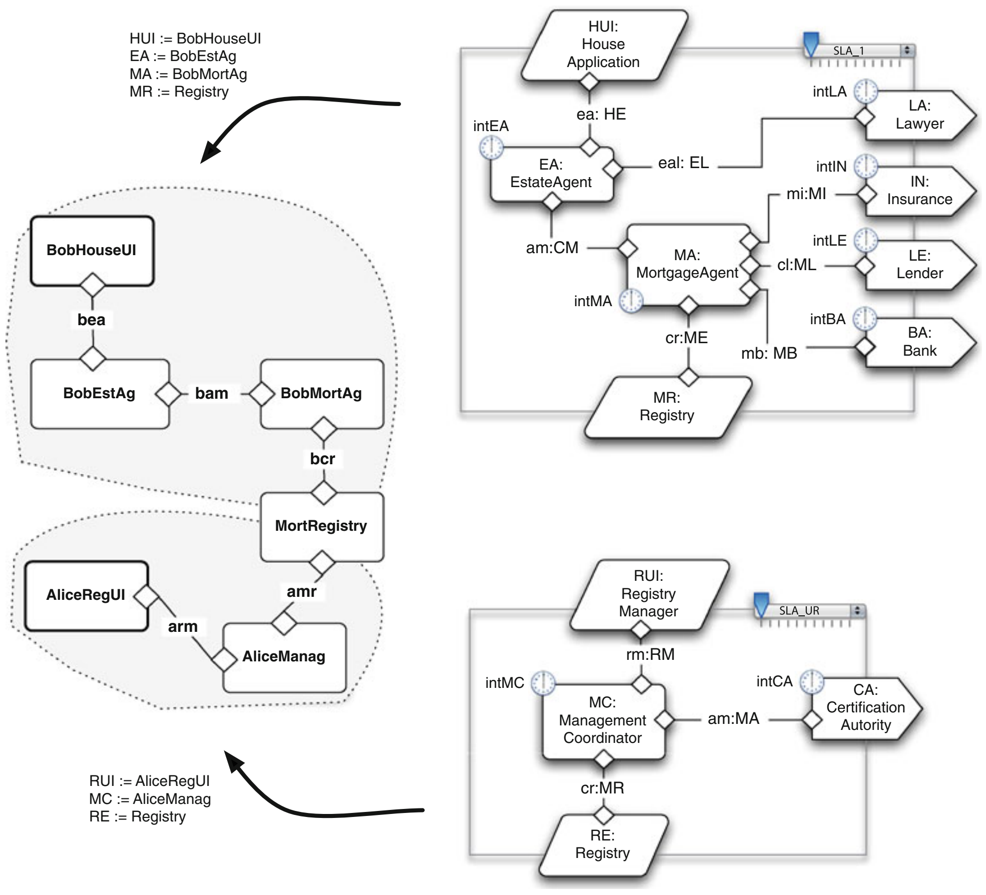

Fig. 3 A business conguration that shows the sub-congurations that correspond to the business activities A_{Bob} (*top part*) and A_{Alice} (*bottom part*) and the activity modules that type them

covery of potential providers of the service and the selection of one service provider among them.

We start by formalising the notion of service, adapting from the model that we developed in SENSORIA [25], which was inspired by concepts proposed in the Service Component Architecture (SCA) [37]. 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 designated component interface through which activities can connect to the service (identified through a requires interface). An additional specification, which we call the *provides-interface* of the service, describes 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.

In order to define the mechanisms of discovery and binding, we need some additional assumptions on the nature of the formalisms that support specification.

Definition 10 (*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 $main(M)$ not in $requires(M) \cup uses(M)$.
– A labelling function SP_M as in definition 8.
– A specification $provides(M)$.
– An internal configuration policy $intPlc(M)$ as in definition 8.
– An external configuration policy $extPlc(M)$ as in definition 8.

The node $main(M)$ represents the component that is responsible for orchestrating the interactions with customers of the service (the formulation of a more general case in which the customer can interact with more than one component can be found in [24]). The specification $provides(M)$ describes the properties that customers can expect from the service. Note that this is not the specification of $main(M)$: the properties offered to customers result from the joint behaviour of the architectural elements defined in the module, including the external services that may need to be discovered, again at run-time.

In order to formalise the relationship between the specification *provides*(M) and the specifications of the architectural elements of the module, we would need to make further assumptions on the structure of the domain **SPEC**. In [24], we define this domain using category theory, more precisely in terms of the category of theory presentations of a π-institution [21] (an abstract notion of an entailment system). For simplicity, we present the case of SRMLight, which is an instance of that more general construction.

Referring to the notion of configuration logic defined in A.3, a service module in SRMLight is said to be *well defined* iff

$$\Phi_{\mathsf{SP}_M} \models provides(M)$$

That is, the properties offered through *provides*(M) derive from the properties of any possible configuration that implements SP_M.

In Fig. 4 we present the structure of the service module that models MortgageFinder—a mortgage brokerage service. A complete definition of this service using the modelling language SRML, including all the specifications involved, is presented in [25]. 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).

The specifications that type interfaces such as *MA*, *RE* and *LE* are the same as those used in the activity module presented in Fig. 2. In what concerns the external policy of the service module, it includes the constraint C_2 presented before but, instead of C_1, it has the following constraint:

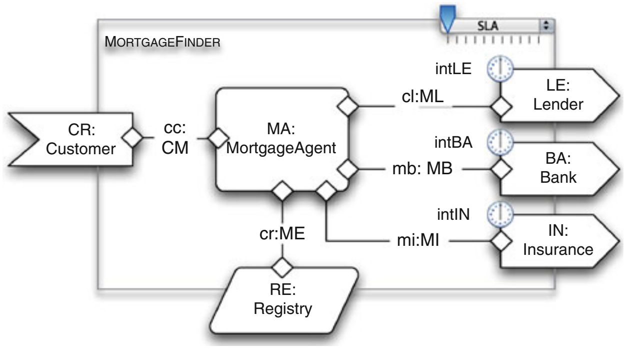

Fig. 4 The structure of a service module that models MortgageFinder

C'_1 : $\{MA.charge,\ MA.replyP.UseBy\}$,

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

This means that, in MortgageFinder, the values of *charge* can be negotiated at the time the service is procured. The constraint defines the negotiation conditions imposed by the service: the higher the *charge* applied to the base price of the brokerage service, the longer the interval during which the proposal is valid.

The provides interface of MortgageFinder is the specification *Customer*. That is, the service module can bind to any activity that requests an external service through a requires-interface that is matched by *Customer*. The signature of the specification *Customer* in SRMLight includes messages that account for requests for a proposal (*reqP*) from a customer and replying to the customer (*replyP*):

V – *charge:nat*
M^+ – *reqP*

P_{reqP} – *usr:usr_data, income:money,*

M^- – *replyP*

P_{replyP} – *prop:mortgage, cost:money*

It also contains the following properties:

$\Box(reqP¡ \supset \Diamond replyP!)$
$\Box([replyP!]$
$\quad replyP.cost = (1 + charge/100) * 750))$

These properties state that the service commits to replying to the request *reqP* by sending the message *replyP*, and that the service brokerage has a base price that is subject to an extra charge, subject to negotiation.

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 the formulation of condition (i) above we assume that the universe **SPEC** of specifications is equipped with a notion of *refinement* such that $\rho : r \to p$ means that the behavioural properties offered by p entail the properties required by r, up to a suitable translation ρ between the languages of both. For example, in SRMLight, a refinement ρ is a mapping between the signatures of the specifications (from the signature of r to the signature of p) such that $p \models \rho(r)$.

The formulation of condition (ii) above relies on a composition operator $\oplus_\rho$ that performs amalgamated unions of constraint systems and sets of constraints, where ρ identifies the variables in both constraint systems that are shared—for more details see [5]. Constraint systems also provide a notion of consistency:

Definition 11 (*Consistency of a set of constraints*) The consistency of a set of constraints is defined in terms of the notion of best level of consistency, which assigns an element of the c-semiring to every set of constraints C as follows:

$$blevel(C) = \sum_t \prod_{c \in C} def_c(t \downarrow con(c))$$

Intuitively, this notion gives us the degree of satisfaction that we can expect for the set of constraints—we choose (through the sum) the best among all possible combinations (product) of all constraints; for more details see [5]. A set of constraints C is said to be consistent iff $blevel(C) >_S 0$. If C is consistent, a valuation for the variables of C is said to be a *solution* for C.

Definition 12 (*Service matching*) Let A be an activity module and $r \in requires(A)$. We denote by **match**$(\mathbf{A}, \mathbf{r})$ the set of pairs $\langle M, \rho \rangle$ such that:

- ρ is a refinement $\mathsf{SP}_A(r) \rightarrow provides(M)$.
- M is a service module such that $sla(M) \oplus_\rho sla(A)$ is consistent.

That is, the matching process for an activity module and one of its requires interfaces returns all the service modules whose provides interface refines the requires interface of the activity module, and whose constraint systems are compatible (in the sense that the refinement identifies which variables of the constraint systems are shared) and whose constraints are consistent.

Definition 13 (*Service discovery*) Let A be an activity module and $r \in requires(A)$. We denote by **discover**$(\mathbf{A}, \mathbf{r})$ the set of triples $\langle M, \rho, \Delta \rangle$ such that:

- $\langle M, \rho \rangle \in$ **match**$(\mathbf{A}, \mathbf{r})$;
- Δ is a solution for $sla(M) \oplus_\rho sla(A)$ such that the degree of satisfaction $bvalue(sla(M) \oplus_\rho sla(A))$ is maximal for **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.

We now define the process of binding $\langle M, \rho, \Delta \rangle$ to a.

Definition 14 (*Service binding*) Let $\mathcal{L} = \langle \mathcal{G}, \mathcal{B}, \mathcal{C} \rangle$ be a business configuration, a an active activity in $\mathcal{L}$, M a service module, $r \in requires(\mathcal{B}(a))$, ρ a refinement mapping from r to $provides(M)$ and Δ a solution. A business configuration $\langle \mathcal{G}', \mathcal{B}', \mathcal{C}' \rangle$ implements the binding of $\langle M, \rho, \Delta \rangle$ to r iff:

- $\mathcal{B}'(a)$ 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 $main(M)$.
  - $requires(M') = requires(M) \cup requires(\mathcal{B}(a)) \setminus \{r\}$, i.e., we add to $\mathcal{B}(a)$ the requires-interfaces of M and eliminate r.
  - $uses(M') = uses(M) \cup uses(\mathcal{B}(a))$, i.e., we add to $\mathcal{B}(a)$ the uses-interfaces of M.
  - $serves(M') = serves(M)$, i.e. we keep the serves-interface of $\mathcal{B}(a)$.
  - the labels $\mathsf{SP}_{M'}$ are those of $\mathcal{B}(a)$ and M applied to the corresponding nodes and edges that remain in M'.
  - $intPlc(M')$ has the triggers and initialisation conditions that are inherited from $\mathcal{B}(a)$ and M.
  - $extPlc(M') =$ $\langle cs(M) \oplus_\rho cs(\mathcal{B}(a)), sla(M) \oplus_\rho sla(\mathcal{B}(a)) \cup \{\Delta\} \rangle$ i.e., we add the solution Δ to the set of constraints inherited from both modules.
- $\mathcal{G}'$ adds to $\mathcal{G}$:
  - For each node $n \in components(M)$, a new component $c_n \in$ **COMP** that implements the specification $\mathsf{SP}_M(n)$ and, for each edge $n \overset{e}{\leftrightarrow} n'$, a wire that implements the specification $\mathsf{SP}_M(e)$.
  - For every node n of $uses(M)$, a component c_n of $\mathcal{G}$ that implements the specification $\mathsf{SP}_M(n)$ and, for every edge $n \overset{e}{\leftrightarrow} n'$, a wire that implements the specification $\mathsf{SP}_M(e)$.

  That is to say, implementations of component-interfaces of M are added to the graph $\mathcal{G}$ and existing components are chosen for uses-interfaces. Wires are added that implement the connectors specified in M.
- $\mathcal{C}'$ 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.

Notice the difference between uses and component interfaces. The former are implemented using components already available in the configuration (thus ensuring persistence), whilst new components (instances) are used as implementations of the latter. Notice also the difference with respect to requires-interfaces, which are not implemented at all: they remain in the business configuration as types so that services that match them can be discovered when (and only when) needed.

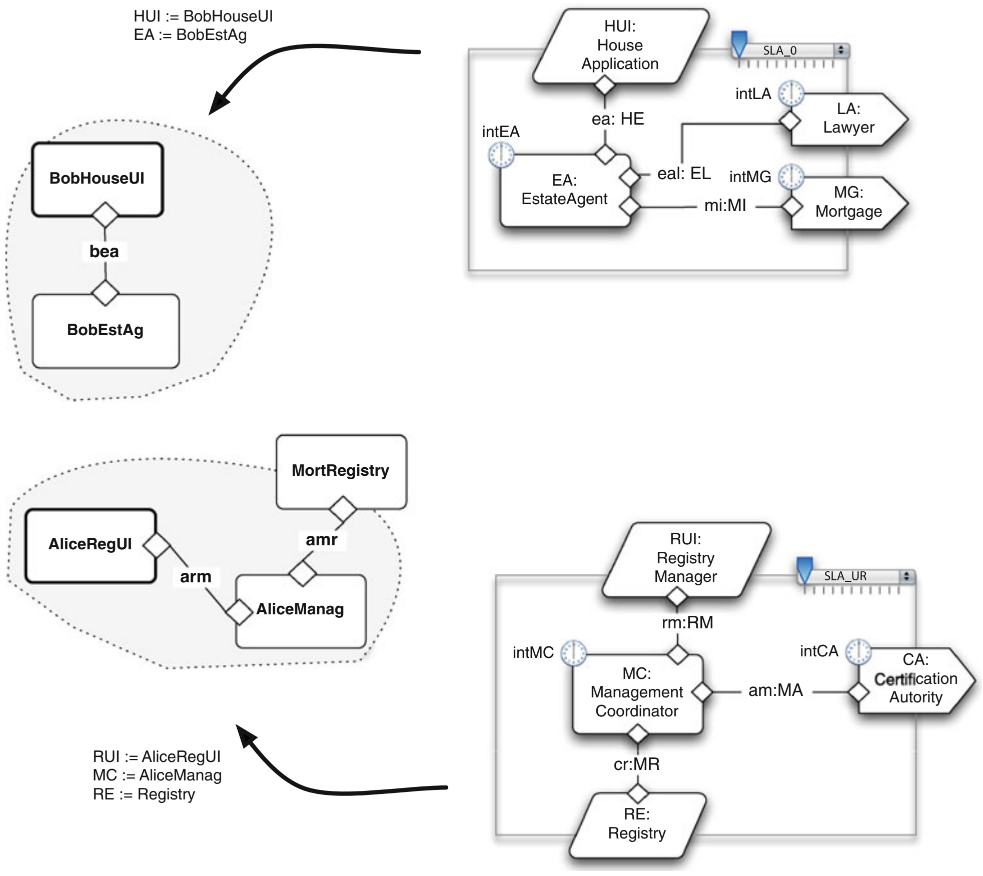

Fig. 5 A business conguration that precedes that of Fig. 3

In order to illustrate how binding works, consider the business configuration in Fig. 5, which shows A_{Bob} at an earlier stage of execution (i.e. earlier than the configuration depicted in the left-hand side of Fig. 1). Assume that, in the current state, the trigger $intMG$ is true and that the service module shown in Fig. 4 is returned by the discovery process described in Definition 13 for the requires-interface MG. A possible result of the binding is depicted in Fig. 3, in which A_{Bob} becomes typed by the activity module in Fig. 2. Notice that the external configuration policy of this new module contains the solution to the constraint C'_1, which is C_1. This explains why we said that C_1 was (so) hard.

Note that a new component—*BobMortAg*—is added to 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 $MortRegistry$. These are 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 example, 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 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.

We can now formalise the notion of reconfiguration step that we discussed in Sect. 2.1.

Definition 15 (*Reconfiguration step*) A business configuration *state* is a quadruple $\langle \mathcal{G}, \mathcal{B}, \mathcal{C}, \mathcal{S}\rangle$ where $\mathcal{L} = \langle \mathcal{G}, \mathcal{B}, \mathcal{C}\rangle$ is a business configuration (cf. Definition 9) and $\langle \mathcal{G}, \mathcal{S}\rangle$ is a state configuration. A reconfiguration *step* consists of a pair of business configuration states $\langle \mathcal{G}, \mathcal{B}, \mathcal{C}, \mathcal{S}\rangle$ and $\langle \mathcal{G}', \mathcal{B}', \mathcal{C}', \mathcal{S}'\rangle$ and a configuration execution step $\mathcal{T}$ between $\mathcal{S}$ and $\mathcal{S}^*$ (as in Definition 3) such that:

- For every activity a in $\mathcal{L}$, $\mathcal{S}'$ coincides with $\mathcal{S}^*$ on $\mathcal{G}(a)$.
- For every activity a in $\mathcal{L}$ and $r \in requires(\mathcal{B}(a))$ such that $\mathcal{T} \models trigger_{\mathcal{B}(a)}(r)$ and $\mathbf{discover}(\mathcal{B}(\mathbf{a}), \mathbf{r}) \neq \emptyset$,
  - $\langle \mathcal{G}', \mathcal{B}', \mathcal{C}'\rangle$ implements the binding of an element $\langle M, \rho, \Delta\rangle$ of $\mathbf{discover}(\mathcal{B}(\mathbf{a}), \mathbf{r})$ to r.
  - For every $n \in components(M)$, $\mathcal{S}'$ assigns to c_n a state that satisfies $init_M(n)$.

– $\mathcal{G}'$ results from $\mathcal{G}$ through the creation of new activities and reconfiguration of existing activities as above.

Notice that every binding extends the current configuration with new components and wires. As no two bindings interfere with each other, several bindings can be performed in just one step.

7 Related work

In the last decade, different approaches to architectural specification have been proposed that permit the representation of dynamic architectures [3,4,14,36,43,44]. 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 [4,14,43], configurator processes [3] or reconfiguration scripts [36,44]. 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 2. 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 [32], π-ADL [34] and ArchWare [33], 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) [16,12,29,31] as well as to capture the dynamic architectures of service-oriented systems [40,35]. 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 number 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 de-referencing 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.

8 Concluding remarks

In this paper we presented a mathematical model that can be used as a semantic domain for architectural description languages that operate over service-oriented architectures. The static aspects of our model were inspired by the concepts proposed in the Service Component Architecture (SCA) [37] 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. This is especially relevant in the absence of standards for these dynamic aspects of SOAs. An example of how an operational account of dynamic reconfiguration under service-oriented architectures can be defined over our mathematical model can be found in [15].

The advantages of having modelling techniques that operate at the more abstract business level have been explored in the language SRML that we defined in SENSORIA [25]. In this paper, we presented a simplified version of SRML—`SRMLight`—and its semantics in order to illustrate how the concepts and constructions that we proposed can be applied to a concrete language. However, 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 [10], which was validated in a number of case studies, including automotive [9] and telco systems [1] in addition to more classical business-oriented domains such as the one used in the paper (a full account of which can be found in [25]).

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 operational verification through model-checking [2], and transformations to PEPA [27] were used for supporting the analysis of quantitative quality-of-service properties such as response time [7]. Those transformations can be supported by tools (prototypes have been developed in MSc projects). The existence of a formal semantic domain such as the one presented in this paper is an essential pre-requisite for transformations to be certified to preserve the semantics of the models that they manipulate, an example of which can be found in [8].

Acknowledgments We would like to thank the reviewers for the suggestions that they gave us to improve the readability of the paper, as well as the reviewers of the conference version of this paper [22] for their comments, and the participants of ECSA 2010 for their helpful feedback. We would also like to thank João Abreu and Laura Bocchi for their collaboration in the definition of SRML and its semantics.

Appendix A: the specification formalisms of `SRMLight`

A.1 Component specifications

The logic defined by a component signature $\langle \mathsf{V}, \mathsf{M} \rangle$ is as follows:

– Its language of actions has the following syntax:

$$\alpha ::= tt \mid m? \mid m¿ \mid m¡ \mid m'! \mid \neg\alpha \mid \alpha \wedge \alpha'$$

where $m \in \mathsf{M}^+$ and $m' \in \mathsf{M}^-$.
Given an execution step t,

$t \models tt$
$t \models m?$ iff $m \in \mathcal{EXC}^t$
$t \models m¿$ iff $m \in \mathcal{PRC}^t \setminus \mathcal{EXC}^t$
$t \models m¡$ iff $m \in \mathcal{DLV}^t$
$t \models m!$ iff $m \in \mathcal{PUB}^t$
$t \models \neg\alpha$ iff not $t \models \alpha$
$t \models \alpha \wedge \alpha'$ iff $t \models \alpha$ and $t \models \alpha'$

– Its language of terms has the following syntax:

$$\beta_d ::= v \mid m.p \mid a \mid f(\beta_{d_1}, \cdots, \beta_{d_n})$$

where $d, d_1, \ldots, d_n \in D$, $v : d \in \mathsf{V}$, $m \in \mathsf{M}$, $p : d \in \mathsf{P}_m$, $a : d \in F$ and $f : d_1 \cdots d_n \to d \in F$.
Given a state s,

$[\![v]\!]_s = \mathcal{VAL}^s(v)$
$[\![m.p]\!]_s = \mathcal{VAL}^s(m, p)$
$[\![a]\!]_s = a_{\mathcal{U}}$
$[\![f(\beta_1, \cdots, \beta_n)]\!]_s = f_{\mathcal{U}}([\![\beta_1]\!]_s, \cdots, [\![\beta_n]\!]_s)$

– Its language of formulas has the following syntax:

$$\phi ::= tt \mid \beta_d = \beta'_d \mid \neg\phi \mid \phi \supset \phi' \mid \bigcirc_\alpha \phi \mid \phi \; {}_\alpha U \; \phi'$$

Given a path $\lambda = \langle s_0 t_0 s_1 t_1 \cdots \rangle$,

$\lambda \models tt$
$\lambda \models (\beta = \beta')$ iff $[\![\beta]\!]_{s_0} = [\![\beta']\!]_{s_0}$
$\lambda \models \neg\phi$ iff not $\lambda \models \phi$
$\lambda \models \phi \supset \phi'$ iff $\lambda \models \phi$ implies that $\lambda \models \phi'$
$\lambda \models \bigcirc_\alpha \phi$ iff $t_0 \models \alpha$ and $\lambda^1 \models \phi$
$\lambda \models \phi \; {}_\alpha U \; \phi'$ iff there exists $0 \leq j$ s.t. $\lambda^j \models \phi'$ and, for all $0 \leq k < j$, $\lambda^k \models \phi$ and $t_k \models \alpha$

– Some useful abbreviations are:

$\underline{\alpha} \equiv \bigcirc_\alpha tt$ — α occurs
$[\alpha]\phi \equiv \neg \bigcirc_\alpha \neg\phi$ — α brings about ϕ
$\Diamond\phi \equiv tt \; {}_{tt}U \; \phi$ — now or eventually ϕ
$\Box\phi \equiv \neg\Diamond(\neg\phi)$ — now and forever ϕ

- For every collection Φ of formulas, $\Lambda_\Phi = \{\lambda : \forall\phi \in \Phi(\lambda \models \phi)\}$. We say that Φ entails ϕ ($\Phi \models \phi$) iff $\Lambda_\Phi \subseteq \Lambda_\phi$. A set Φ is consistent iff $\Lambda_\Phi \neq \emptyset$.
- A *specification* is a collection Φ of formulas that is consistent. A *model* of a specification Φ is an element (path) of Λ_Φ.

Typical specifications include formulas of the form:

- $\Box(guard \supset [m?]effects)$—the execution of m when *guard* holds brings about *effects*
- $\Box(guard \supset \neg m¿)$—$m$ will not be discarded when $guard$ holds
- $\Box([request¡]\Diamond reply!)$—the delivery of *request* ensures the publication of *reply*

A.2 Wire and connection specifications

The logic of an A-wire signature M is defined in the same way as that of a component signature with an empty set of variables. The logic defined by an A-connection ($c_1 \overset{\mu_1}{\leftarrow} w \overset{\mu_2}{\rightarrow} c_2$) is as follows:

- Its language of actions has the following syntax:

  $$\alpha ::= tt \mid c_j.\mu_j(m)¡ \mid c_i.\mu_i(m)! \mid \neg\alpha \mid \alpha \wedge \alpha'$$

  where $m \in \mu_i^{-1}(\mathsf{M}_{c_i}^-)$, $\{i, j\} = \{1, 2\}$.
  Given an execution step t,

  $$\begin{aligned} &t \models tt \\ &t \models c_j.\mu_j(m)¡ \text{ iff } m \in \mathcal{DLV}^t \\ &t \models c_i.\mu_i(m)! \text{ iff } m \in \mathcal{PUB}^t \\ &t \models \neg\alpha \text{ iff not } t \models \alpha \\ &t \models \alpha \wedge \alpha' \text{ iff } t \models \alpha \text{ and } t \models \alpha' \end{aligned}$$

- Its language of terms has the following syntax:

  $$\beta_d ::= c_i.\mu_i(m).p \mid a \mid f(\beta_{d_1}, \cdots, \beta_{d_n})$$

  where $d, d_1, \ldots, d_n \in D$, $a : d \in F$ and $f : d_1 \cdots d_n \to d \in F$, $m \in \mathsf{M}$ and $p : d \in \mathsf{P}_{\mu_i(m)}$.
  Given a state s,

  $$\begin{aligned} &[\![c_i.\mu_i(m).p]\!]_s = \mathcal{VAL}^s_{c_i}(\mu_i(m), p) \\ &[\![a]\!]_s = a_\mathcal{U} \\ &[\![f(\beta_1, \cdots, \beta_n)]\!]_s = f_\mathcal{U}([\![\beta_1]\!]_s, \cdots, [\![\beta_n]\!]_s) \end{aligned}$$

- Its language of formulas is as for component signatures.
- A *specification* is a collection Φ of formulas that is consistent.

That is, the logic of A-connections uses the messages (and corresponding parameters) of the components being connected. We prefix them with the (name of) the corresponding components so as to avoid name clashes—as already mentioned, different components may use the same names for messages (or variables).

The logic defined by an S-connection ($c_1 \overset{\mu_1}{\rightarrow} w \overset{\mu_2}{\leftarrow} c_2$) is as follows:

- Its language of terms has the following syntax ($i = 1, 2$):

  $$\beta_d ::= c_i.\mu_i^{-1}(v) \mid a \mid f(\beta_{d_1}, \cdots, \beta_{d_n})$$

  where $d, d_1, \cdots, d_n \in D$, $f : d_1 \cdots d_n \to d \in F$, $v : d \in \mathsf{V}$, $a : d \in F$.
  Given a state s,

  $$\begin{aligned} &[\![c_i.\mu_i^{-1}(v)]\!]_s = \mathcal{VAL}^s_{c_i}(\mu_i^{-1}(v)) \\ &[\![a]\!]_s = a_\mathcal{U} \\ &[\![f(\beta_1, \cdots, \beta_n)]\!]_s = f_\mathcal{U}([\![\beta_1]\!]_s, \cdots, [\![\beta_n]\!]_s) \end{aligned}$$

- Its language of formulas has the following syntax:

  $$\phi ::= \beta_d = \beta'_d \mid \Box\phi$$

  interpreted as for component signatures.
- A *specification* is a collection Φ of formulas that is consistent.

A.3 Logic for configurations

The relationship between specifications and architectural elements is formalised in terms of execution models as follows: let c be a component and Φ a component specification with the same signature; we say that c implements Φ iff $\Lambda_c \subseteq \Lambda_\Phi$; the same applies to connections (wires). That is, a component (connection) implements a specification if all the execution paths of the component (connection) satisfy the specification.

A specification for a configuration $\mathcal{G}$ is a mapping SP that assigns a specification to every component and connection such that every architectural element implements the corresponding specification.

In order to reason about the global properties of a configuration, it is useful to define a logic for the configuration itself. In SRMLight, this logic is defined as follows.

The signature of a configuration specification SP with domain $\mathcal{G}$ is the pair $\langle \mathsf{V}, \mathsf{M} \rangle$ where:

- $\mathsf{V} = \bigcup_{c \in nodes(\mathcal{G})} c.\mathsf{V}_c$
- $\mathsf{M}^+ = \bigcup_{c \in nodes(\mathcal{G})} c.\mathsf{M}_c^+$.
- $\mathsf{M}^- = \bigcup_{c \in nodes(\mathcal{G})} c.\mathsf{M}_c^-$.

where $\langle \mathsf{V}_c, \mathsf{M}_c \rangle$ is the signature of the component c and $c.\mathsf{V}_c$ (resp. $c.\mathsf{M}_c^-$ and $c.\mathsf{M}_c^+$) is the result of prefixing the variables of V_c (resp. messages of M_c^-, M_c^+) with c.

The language associated with this signature is as for component signatures. This language is interpreted over configuration states and steps as follows:

– $\mathcal{T} \models c.m?$ iff $\mathcal{T}_c \models m?$, idem for $m¿$, $m¡$ and $m!$
– $[\![c.v]\!]_{\mathcal{S}} = [\![v]\!]_{\mathcal{S}_c}$

The theory presentation associated with the configuration specification SP, which we denote by Φ_{SP}, is the union of the following sets of formulas:

– For every component c, the translation $c.\mathsf{SP}_c$, which is obtained by replacing every variable and message with the prefix c.
– For every wire, the specification of its connection.

The set Φ_{SP} is the union of the translations of the specifications of all components and wires. The following theorem establishes that this set provides a specification for the execution paths of the configuration.

Theorem 16 *For every specification SP for a configuration $\mathcal{G}$,*

$\Lambda_{\mathcal{G}} \subseteq \Lambda_{\Phi_{\mathsf{SP}}}$

Proof (sketch) Because SP is such that every component and connection implements the corresponding specification, i.e., $\Lambda_c \subseteq \Lambda_\Phi$ for every component c (and mutatis mutandis for wires), and every execution path in $\Lambda_{\mathcal{G}}$ projects to Λ_c for every component c (and mutatis mutandis for wires)—see Definition 5—all such execution paths satisfy the set of formulas $\Lambda_{\Phi_{\mathsf{SP}}}$, i.e. the union of the translations of the specifications Λ_c (mutatis mutandis for wires). □

A.4 State and trigger conditions

State conditions are of the form

$\phi ::= tt \mid \beta_d = \beta'_d \mid \neg\phi \mid \phi \supset \phi'$

with

$\beta_d ::= v \mid m.p \mid a \mid f(\beta_{d_1}, \ldots, \beta_{d_n})$

where $d, d_1, \ldots, d_n \in D$, $f : d_1 \cdots d_n \rightarrow d \in F$, $v : d \in \mathsf{V}$ and $a : d \in F$.

Triggers are pairs $\langle \phi, m¡ \rangle$ where ϕ is as above. The delivery of the message is evaluated during the transition and the condition is evaluated on the state before the transition.

References

1. Abreu, J., Bocchi, L., Fiadeiro, J.L., Lopes, A.: Specifying and composing interaction protocols for service-oriented system modelling. In: Derrick, J., Vain, J. FORTE. LNCS, vol. 4574, pp. 358–373. Springer, Berlin (2007)
2. Abreu, J., Mazzanti, F., Fiadeiro, J.L., Gnesi, S.: A model-checking approach for service component architectures. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. FMOODS/FORTE. LNCS, vol. 5522, pp. 219–224. Springer, Berlin (2009)
3. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: FASE, pp. 21–37 (1998)
4. Batista, T.V., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In: Morrison, R., Oquendo, F. EWSA. LNCS, vol. 3527, pp. 1–17. Springer, Berlin (2005)
5. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. J. ACM **44**(2), 201–236 (1997)
6. Bistarelli, S., Santini, F.: A nonmonotonic soft concurrent constraint language for SLA negotiation. Electron. Notes Theor. Comput. Sci **236**, 147–162 (2009)
7. Bocchi, L., Fiadeiro, J.L., Gilmore, S., Abreu, J., Solanki, M., Vankayala, V.: A formal approach to modelling time properties of service oriented systems. In: Handbook of Research on Non-Functional Properties for Service-Oriented Systems: Future Directions. Advances in Knowledge Management Book Series. IGI Global (2012, in press)
8. Bocchi, L., Fiadeiro, J.L., Lapadula, A., Pugliese, R., Tiezzi, F.: From architectural to behavioural specification of services. Electron. Notes Theor. Comput. Sci **253**(1), 3–21 (2009)
9. Bocchi L., Fiadeiro J.L., Lopes A.: Service-oriented modelling of automotive systems. In: COMPSAC. IEEE Computer Society, pp. 1059–1064 (2008)
10. Bocchi, L., Fiadeiro, J.L., Lopes, A.: A use-case driven approach to formal service-oriented modelling. In: Margaria, T., Steffen, B. ISoLA. Communications in Computer and Information Science, vol. 17, pp. 155–169. Springer, Berlin (2008)
11. Bocchi, L., Hong, Y., Lopes, A., Fiadeiro, J.L.: From BPEL to SRML: A formal transformational approach. In: Dumas, M., Heckel, R. WS-FM. LNCS, vol. 4937, pp. 92–107. Springer, Berlin (2007)
12. Boreale, M. et al: SCC: a service centered calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. WS-FM. LNCS, vol. 4184, pp. 38–57. Springer, Berlin (2006)
13. Broy, M.: From "formal methods" to system modeling. In: Jones, C.B., Liu, C.B., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. Lecture Notes in Computer Science, vol. 4700, pp. 24–44. Springer, Berlin (2007)
14. Bruni, R., Bucchiarone, A., Gnesi, S., Hirsch, D., Lluch-Lafuente, A.: Graph-based design and analysis of dynamic software architectures. In: Degano, P., Nicola, R.D., Meseguer, J. Concurrency Graphs and Models. LNCS, vol. 5065, pp. 37–56. Springer, Berlin (2008)
15. Bruni, R., Lluch-Lafuente, A., Montanari, U., Tuosto, E.: Service oriented architectural design. In: Barthe,G., Fournet, G. (ed.) TGC. Lecture Notes in Computer Science, vol. 4912, pp. 186–203. Springer, Berlin (2007)
16. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: De Nicola [18], pp. 2–17
17. Coulson, G., Blair, G.S., Grace, P., Taïani, F., Joolia, A., Lee, K., Ueyama, J., Sivaharan, T.: A generic component model for building systems software. ACM Trans. Comput. Syst. **26**(1), (2008)
18. De Nicola, R. (ed.): Programming Languages and Systems. LNCS, vol. 4421. Springer, Berlin (2007)

19. Elfatatry, A.: Dealing with change: components versus services. Commun. ACM **50**(8), 35–39 (2007)
20. Fargier, H., Lang, J., Schiex, T.: Mixed constraint satisfaction: a framework for decision problems under incomplete knowledge. In: AAAI/IAAI, vol. 1, pp. 175–180 (1996)
21. Fiadeiro, J.L.: Categories for Software Engineering. Springer, Berlin (2004)
22. Fiadeiro, J.L., Lopes, A.: A model for dynamic reconfiguration in service-oriented architectures. In: Babar, M.A., Gorton, I. ECSA. LNCS, vol. 6285, pp. 70–85. Springer, Berlin (2010)
23. Fiadeiro, J.L., Lopes, A., Abreu, J.: A formal model for service-oriented interactions. Sci. Comput. Program (2011, in print)
24. Fiadeiro, J.L., Lopes, A., Bocchi, L.: An abstract model of service discovery and binding. Formal Asp. Comput. **23**(4), 433–463 (2011)
25. Fiadeiro, J.L., Lopes, A., Bocchi, L., Abreu, J.: The SENSORIA reference modelling language. In: Wirsing and Hölzl [45], pp. 61–114
26. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B.R., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. IEEE Comput. **37**(10), 46–54 (2004)
27. Gilmore, S., Hillston, J.: The PEPA workbench: a tool to support a process algebra-based approach to performance modelling. In: Haring, G., Kotsis, G. Computer Performance Evaluation. LNCS, vol. 794, pp. 353–368. Springer, Berlin (1994)
28. Kon, F., Costa, F.M., Blair, G.S., Campbell, R.H.: The case for reflective middleware. Commun. ACM **45**(6), 33–38 (2002)
29. Lapadula, A., Pugliese, R., Tiezzi F.: A calculus for orchestration of web services. In: De Nicola [18], pp. 33–47
30. Léger, M., Ledoux, T., Coupaye, T.: Reliable dynamic reconfigurations in a reflective component model. In: Grunske, L., Reussner, R., Plasil, F. CBSE. LNCS, vol. 6092, pp. 74–92. Springer, Berlin (2010)
31. Lucchi, R., Mazzara, M.: A π-calculus based semantics for WS-BPEL. J. Log. Algebr. Program. **70**(1), 96–118 (2007)
32. Magee, J., Kramer J.: Dynamic structure in software architectures. In: SIGSOFT FSE, pp. 3–14 (1996)
33. Morrison, R., Kirby, G.N.C., Balasubramaniam, D., Mickan, K., Oquendo, F., Cîmpan, S., Warboys, B., Snowdon, B., Greenwood, R.M.: Support for evolving software architectures in the archware ADL. In: WICSA, pp. 69–78. IEEE Computer Society (2004)
34. Oquendo, F.: π-ADL: an architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT Softw. Eng. Notes **29**(3), 1–14 (2004)
35. Oquendo, F. Formal approach for the development of business processes in terms of service-oriented architectures using π-ADL. In: Lee, J., Liang, D., Cheng, Y.C. SOSE, pp. 154–159. IEEE Computer Society, Berlin (2008)
36. Oreizy, P., Taylor, R.N.: On the role of software architectures in runtime system reconfiguration. IEE Proc. Softw. **145**(5), 137–145 (1998)
37. OSOA. Service component architecture: Building systems using a service oriented architecture, 2005. White paper available from http://www.osoa.org
38. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes **17**(4), 40–52 (1992)
39. Reichel, H.: Initial Computability, Algebraic Specifications, and Partial Algebras. Oxford University Press Inc., New York (1987)
40. Sanz, M.L., Qayyum, Z., Cuesta, C.E., Marcos, E., Oquendo, F.: Representing service-oriented architectural models using π-ADL. In: Morrison, R., Balasubramaniam, D., Falkner, K.E. ECSA. LNCS, vol. 5292, pp. 273–280. Springer, Berlin (2008)
41. Simonot, M., Aponte V.: A declarative formal approach to dynamic reconfiguration. In: Proceedings of the 1st international workshop on Open component ecosystems, IWOCE'09, pp. 1–10. ACM, New York, 2009
42. Tiberghien, A., Merle, P., Seinturier, L.: Specifying self-configurable component-based systems with fractoy. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. ASM. LNCS, vol. 5977, pp. 91–104. Springer, Berlin (2010)
43. Wermelinger, M., Fiadeiro, J.L.: A graph transformation approach to software architecture reconfiguration. Sci. Comput. Program. **44**(2), 133–155 (2002)
44. Wermelinger, M., Lopes, A., Fiadeiro, J.L.: A graph based architectural (re)configuration language. In: ESEC / SIGSOFT FSE, pp. 21–32 (2001)
45. Wirsing, M., Hölzl, M. (eds.): Rigorous Software Engineering for Service-Oriented Systems. LNCS, vol. 6582. Springer, Berlin (2011)

Author Biographies

José Luiz Fiadeiro is Professor for Software Science and Engineering at the University of Leicester, which he joined in 2002 after having held academic positions at the Technical University of Lisbon and the University of Lisbon. He also held visiting positions at Imperial College London, King's College London, PUC-Rio de Janeiro (Brazil), and the SRI International (California). He was Head of Department at Leicester from August 2006 to July 2011. José's current research interests are in formal aspects of software system modelling and analysis in the context of global ubiquitous computing. He is also-member of the Editorial Board of Information Processing Letters (Elsevier) and Fellow of the British Computer Society.

Antónia Lopes is Associate Professor at the University of Lisbon, Portugal, since March 2006. She received a Ph.D. in Informatics at the University of Lisbon in 1999. Her research interests are in design principles, theories and techniques that support the modelling and analysis of various types of software intensive systems, namely service-oriented systems and self-adaptive systems.