

# An Abstract Model of Service Discovery and Binding

José Luiz Fiadeiro<sup>1</sup>, Antónia Lopes<sup>2</sup> and Laura Bocchi<sup>1</sup>

<sup>1</sup>Department of Computer Science, University of Leicester, UK

<sup>2</sup>Department of Informatics, Faculty of Sciences, University of Lisbon, PORTUGAL

**Abstract.** We propose a formal operational semantics for service discovery and binding. This semantics is based on a graph-based representation of the configuration of global computers typed by business activities. Business activities execute distributed workflows that can trigger, at run time, the discovery, ranking and selection of services to which they bind, thus reconfiguring the workflows that they execute. Discovery, ranking and selection are based on compliance with required business and interaction protocols and optimisation of quality-of-service constraints. Binding and reconfiguration are captured as algebraic operations on configuration graphs. We also discuss the methodological implications that this model framework has on software engineering using a typical travel-booking scenario. To the best of our knowledge, our approach is the first to provide a clear separation between service computation and discovery/instantiation/binding, and to offer a formal framework that is independent of the SOA middleware components that act as service registries or brokers, and the protocols through which bindings and invocations are performed.

**Keywords:** Business-reflective run-time configurations; Dynamic reconfiguration; Quality-of-service constraints; Service binding, discovery, ranking; Service level agreement; Service-oriented computing

## 1. Introduction

Service-Oriented Computing (SOC) is a new paradigm in which interactions are no longer based on the exchange of products with specific parties — what is known as clientship in object-oriented programming — but on the provisioning of services by external providers that can be procured on the fly subject to a negotiation of service level agreements (SLAs). A number of research initiatives (among them the FET-GC2 integrated project SENSORIA [WHar]) have been proposing formal approaches that address different aspects of the paradigm independently of the specific languages that are available today for Web Services (e.g., [BMR97, Pel03]) or Grid Computing (e.g., [FK04]). For example, recent proposals for service calculi (e.g., [BBC<sup>+</sup>06, CHY07, LPT07, KQCM09]) address operational foundations of SOC (in the sense of how services compute) by providing a mathematical semantics for the mechanisms that support choreography or orchestration — sessions, message/event correlation, compensation, *inter alia*. This line of work has

---

*Correspondence and offprint requests to:* J. Fiadeiro, Department of Computer Science, University of Leicester, University Road, Leicester LE1 7RH, UK, e-mail: jose@mcs.le.ac.uk

contributed to languages and standards developed by organisations such as OASIS ([www.oasis-open.org](http://www.oasis-open.org)) and W3C ([www.w3.org](http://www.w3.org)) for Web Services.

Whereas such calculi address the need for specialised language primitives for programming in this new paradigm, we are still lacking models that are abstract enough to understand the engineering foundations of SOC, i.e., those aspects (both technical and methodological) that concern the way applications can be developed to provide business solutions, independently of the languages in which services are programmed. The Open Service Oriented Architecture collaboration ([www.osoa.org](http://www.osoa.org)) has been proposing a number of specifications, namely the Service Component Architecture (SCA), that address this challenge: “*SCA aims to provide a model for the creation of service components in a wide range of languages and a model for assembling service components into a business solution — activities which are at the heart of building applications using a service-oriented architecture*” [OSO05].

However, SCA addresses low-level design in the sense that it provides an assembly model and binding mechanisms for service components and clients programmed in specific languages, e.g., Java, C++, BPEL, or PHP. The goal of the work that we discuss in this paper is to address high-level design. More specifically, we aim for models and mechanisms that support the design of complex services from business requirements in ways that are independent of the languages in which the service components are programmed, and for analysis techniques through which designers can verify or validate properties of composite services. So far, SOC has been short of support for high-level modelling. Indeed, languages and models that have been proposed for service modelling and design (e.g., [BKM07, DD04, Rei05]) do not address the higher level of abstraction that is associated with business solutions, in particular the key characteristic aspects of SOC that relate to the way those solutions are put together dynamically in reaction to the execution of business processes — run-time discovery, instantiation and binding of services. Yet, these dynamic aspects are among those that truly distinguish SOC from component-based development and other forms of distributed computing [Elf07, GL07]. They are also the aspects that are particularly relevant for the engineering foundations of SOC.

This is why, within SENSORIA, we have defined the modelling language SRML [FLBAar]. Its ‘static’ aspects — the design-time definition of complex services in terms of orchestrations of simpler services — are formalised over state transition systems and temporal logic [AF08, FLA10]. Its ‘dynamic’ aspects are formalised over the mathematical model put forward in this paper. More specifically, the model that we propose herein is, essentially, parametric in the formalisms adopted for modelling the static aspects and, therefore, can be instantiated for those used in SRML. To the best of our knowledge, our approach is the first to provide a clear separation between service computation and discovery/instantiation/binding, and to offer a formal framework that is independent of the SOA middleware components that act as service registries or brokers (e.g., the UDDI [UDD04]) and the protocols through which bindings and invocations are performed (e.g., SOAP [W3C07]).

In the rest of this section, we make more precise the nature and the role of the mathematical model that we define in the paper. We also present the example that will be used to illustrate the way our model supports SOC. In Section 2, we expand on the notion of service-overlay computer; we discuss the notion of a service that we address in the paper and the impact of SOC on software engineering methodology; then, we put forward a layered graph-based model for state configurations of global computers, made to be ‘business reflective’ by enriching them with an explicit representation of the types of business activities that are active in the current state. In Section 3, we discuss the typing mechanism that we propose for services and a notion of correctness that acts as an abstract static semantics of services; we also formalise the notion of quality of service. In Section 4, we present a formal model for service discovery and binding. In Section 5, we compare our proposal to other related approaches. We conclude with an overview of the way the proposed formal model was used within SENSORIA in support of a service-based modelling approach to software-intensive systems.

## 1.1. Models for service-oriented computing

Mathematical models that capture the essence of a paradigm at the desired level of abstraction play an essential role as a foundation for methods, languages and support tools for that paradigm. The choice of a model reflects the level of abstraction at which one wishes to capture a particular aspect of a paradigm. In this paper, our concern is not the behaviour of software applications viewed in terms of the transformations that they perform over data or the events that they exchange with their environment. Like in SCA, we

address the coarser-grain process that SOC induces over the configuration of systems because of the ability that applications have to bind dynamically to services that are discovered, selected and instantiated at run time. This is why, drawing an analogy with the semantics of programming languages, we could say that we put forward mathematical notions of (typed) state and state transition that account for the evolutionary process that SOC induces over software systems.

Therefore, the proposed model addresses the layer of abstraction that SOAs superpose over any concrete computer platform in terms of service discovery, instantiation and binding. Just like developers working over an object-oriented platform do not need to program the dynamic allocation, referencing and de-referencing of names, designers of complex services working over a SOA should not need to include the discovery, selection and binding processes among the tasks of the orchestrators of the services.

In this sense, our aim is to provide an abstract virtual machine that captures the essence of SOC as a ‘business overlay computer’. Our operational model does not include the service broker or the service providers as components of the system — it relies on them as resources available in the architecture in the same way as an operational model for sequential programming does not include memory allocation or access as processes. The availability of abstract models such as the one we propose in this paper is a precondition for defining semantically-enhanced service discovery [PTDL07], a challenge that has already led to the definition of a run-time infrastructure consisting of a number of brokers that handle quality-of-service constraints [MDSR07].

More specifically, in the model that we propose in Section 2, the states of the abstract virtual machine are graphs of components and connectors that capture configurations that execute business activities, and state transitions are reconfigurations that result from binding to selected services. In order to make the model reflective (in the sense of [CBG<sup>+</sup>08], as discussed in Section 2.3), states are typed with so-called activity modules, which determine the state transitions that are possible, i.e., the services that need to be discovered and the criteria for selection — matching required functional properties and optimising quality-of-service constraints. In Section 2.2, we also discuss how these abstract states and transitions of the configuration process relate to the computational model that captures the way components execute in each configuration.

## 1.2. The travel-booking scenario

In order to illustrate our approach, we use a travel-booking scenario that goes somewhat beyond what is typically found in papers on SOA. In this example, we consider the case of a company that wishes to offer as a service the ability to book a flight and a hotel according to given preferences, and handle the corresponding payments. For that purpose, the company has a database of clients and an agent that orchestrates a booking process that relies on three external agents: one that handles the booking of the flights, another the booking of the hotel, and a third the payments. In order to be able to offer the best possible deal at any moment and take into account data of the specific customer, these external agents are procured at run time subject to given service-level agreements.

From the software development point of view, adopting a service-oriented approach means that the booking agent is not programmed to perform the discovery and selection of these external agents. These tasks are left to the middleware, which is also responsible for binding the booking agent with the selected agents. From a modelling point of view, this means that the task of the designer is just to define the criteria according to which the discovery and selection should be made. Concerning the database of users, although it is external to the service, it is internal to the service provider and does not need to be discovered — binding takes place when the service is invoked and the booking agent is instantiated.

The customers of the service offered by this company will be software components that implement business activities run by other companies, which require the use of a travel agent. This means that, in order to be discovered by these activities, our travel-booking service provider needs to register with a directory in which it advertises the properties of the service that it offers. Typically, these business activities will include an interface to a user that can invoke them directly. In our example, this user could be a person wishing to go on holiday who uses a web browser to launch an activity on their favourite holiday agent’s site. That activity would then execute its business process over a SOA, which could include binding dynamically to a travel agent that, say, offers a best match for the user’s environmental concerns or to a trusted agency that can provide a nanny to look after the user’s children.

## 2. Business-Reflective Configurations of Global Computers

### 2.1. Service-overlay engineering methodology

The term ‘service’ is being used in Information Technology (IT) in a wide variety of contexts, often with different meanings. In this paper, we address the notion of ‘service-overlay computer’, by which we mean the development of highly-distributed loosely-coupled applications over ‘global computers’ — “*computational infrastructures available globally and able to provide uniform services with variable guarantees for communication, co-operation and mobility, resource usage, security policies and mechanisms*” (see the Global Computing Initiative at [cordis.europa.eu/ist/fet/gc.htm](http://cordis.europa.eu/ist/fet/gc.htm)).

In this context, it is necessary to rethink the way software applications are engineered. It is clear that the typical ‘static’ scenario in which components are assembled to build a (more or less complex) system that is delivered to a customer no longer applies. Instead, SOC supports more ‘dynamic’ development scenarios in which (smaller) applications are developed to run on global computers and respond to business needs by interacting with services and resources that are globally available. In this latter setting, there is much more scope for flexibility in the way business is supported: business processes need not be confined to fixed organisational contexts; they can be viewed in more global contexts as emerging from a varying collection of loosely coupled applications that can take advantage of the availability of services procured on the fly when they are needed.

Indeed, an important impact that we see SOC having on software engineering methodology derives from the fact that the discovery and selection of services required by business applications is performed, on the fly, by the middleware (not by skilled engineers at design time), and that this selection is not made from a fixed universe of business components but an open and dynamic market of service provision. Likewise, developing services that can be discovered is not the same as developing software applications to a customer’s set of requirements: it is a separate business that exists independently of the nature of the customers. This independence requires the definition of shared ontologies of data and services so that providers and requesters can negotiate and agree on levels of service provision.

This view is summarised (in a somewhat abstract form) in Figure 1, where we elaborate beyond the basic Service-Oriented Architecture [ACKM04] to make explicit the different stakeholders and the way they interact, which is important for understanding the formal model that we are proposing. In this model, we distinguish between ‘business activities’ and ‘services’ as software applications that pertain to different stakeholders (see [GL07] for a discussion on the stakeholders of service-oriented systems):

- *Activities* correspond to applications developed by ‘business IT teams’ according to requirements provided by their organisation, e.g., the applications that, in a tour operator, implement the products that are made available to its customers. The activity repository provides a means for the computing infrastructure of the organisation to trigger such applications when the corresponding requests are published, say when a client requests a quote for a holiday at a counter or through the web. The implementation of activities may resort to ‘classical’ direct invocation of components (e.g., for exchange rates or medical insurance), but it can also rely on services that will be procured on the fly (for instance, an agent that can take care of the travel requirements) to ensure competitiveness and context-based optimisation.
- *Services* differ from activities in that they are not developed to satisfy specific business requirements of a given organisation. Instead they are developed by service providers who publish them (in service repositories) in ways that allow them to be discovered when a request for an external service is published in the run-time environment. As such, they are classified according to generic service descriptions that are organised in a hierarchical ontology to facilitate discovery. In the SOA middleware that is currently available, the UDDI [UDD04] is an example of a (less sophisticated) service registry or broker.

Notice that the ‘business IT teams’ and the ‘service providers’ can be totally independent and unrelated: the former are interested in supporting the business of their companies or organisations, whereas the latter run their own businesses. They share the ontology component of the architecture so that they can do business together. In addition to these two repositories, our engineering infrastructure includes:

- A configuration management unit, which is responsible for the binding of the new components and connectors that derive from the instantiation of new activities or services. Current SOA middleware uses mechanisms like SOAP [W3C07] for providing the protocols through which bindings and invocations are performed.

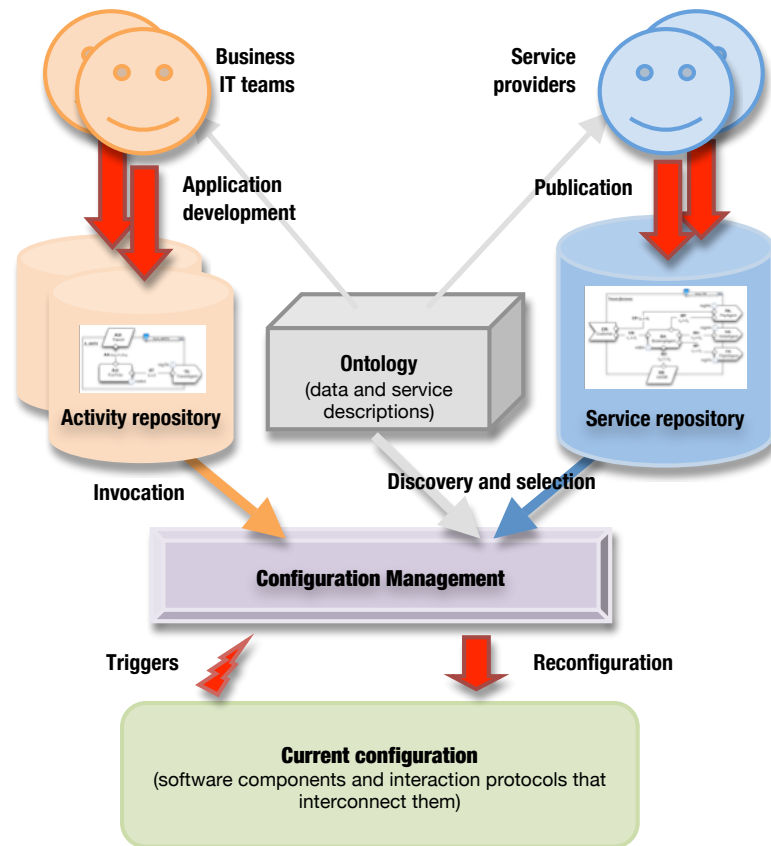


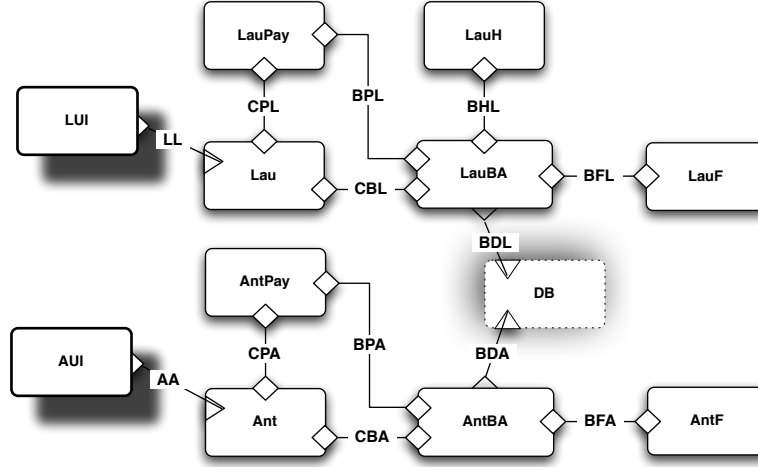
Fig. 1. Overall engineering architecture and processes

- An ontology unit, which is responsible for organising both data and service descriptions. This is an area that is still lacking standards, though there has been substantial progress in the development of Semantic Web techniques (see the W3C Semantic Web Activity at [www.w3.org/2001/sw/](http://www.w3.org/2001/sw/)).

In the context of this infrastructure, the paper makes the following contributions:

- It proposes a formal (graph-based) model for the ‘current configuration’ (Section 2.2).
- It proposes formal notions of type for business activities and services that are parametric on the formalisms that are used for describing orchestrations and interactions (Sections 2.3 and 3.1).
- It uses these types to make the current configuration business-reflective, i.e., it adds to the configuration a level of representation that makes the underlying business model explicit and uses that representation for rewriting the configuration according to given business rules (Sections 2.3 and 3.1).
- It proposes a formal notion of SLA and negotiation based on constraint optimisation with c-semirings [BMR97] (Sections 3.2 and 4.1).
- It proposes a formal model of service discovery and binding over such business-reflective configurations (Section 4.1).

We do not discuss the classification and retrieval mechanisms per se, i.e., the processes through which the activity and service repositories are managed. See, for instance, [Pah07, RS04] for some of the aspects involved when addressing such issues.



**Fig. 2.** The graph of a state configuration with 12 components and 13 wires decorated with the iconography of SRML (summarised in the Appendix)

## 2.2. Layered state configurations of global computers

As already mentioned, we take SOC to be about applications that can bind to other applications discovered at run time in a universe of resources that is not fixed a priori. As a result, there is no structure or configuration (or what sometimes is called ‘system architecture’) that one can fix at design-time for an application; rather, there is an underlying notion of configuration of a global computer that keeps being redefined as applications execute and get bound to other applications that offer required services. As is often the case (e.g., [AG98]), by ‘configuration’ we mean a graph of components (applications deployed over a given execution platform) linked through wires (e.g., interconnections between components over a given communication network) in a given state of execution. Typically, wires deal with the heterogeneity of the partners involved in the provision of the service, performing data (or, more, generally, semantic) integration.

Figure 2 presents an example of a configuration related to the case study described in Section 1.2 — the nodes of the configuration graph correspond to the boxes and the edges to the lines (the meaning of the shadows is explained below). The example consists of two instances of a travel-booking activity sharing a database DB of users. Each of these activities corresponds to a sub-graph that captures a distributed orchestration of the activity, as explained in Section 2.3 and illustrated in Figure 3.

We denote by  $\text{COMP}$  and  $\text{WIRE}$  the set of all components and wires, respectively. Every component  $c \in \text{COMP}$  and wire  $w \in \text{WIRE}$  may be in a number of states (e.g., valuations of local state variables), the set of which is denoted by  $\text{STATE}_c$  and  $\text{STATE}_w$ , respectively. We denote by  $\text{STATE}$  the corresponding indexed family of sets of states. The precise nature of these local states is of no particular importance for this paper — in relation to the semantics of discovery and binding, the dependency on states concerns the evaluation of the conditions that trigger the discovery of external services (as discussed in Section 4.1) and the initialisation of the new components and wires that result from the binding (as discussed in Section 4.2), both of which are dealt with what we call ‘internal configuration policies’ (see Section 3.2). Consequently, we refrain from making any further assumptions on  $\text{STATE}$ . The specific nature of the states and state transitions used in SRML is presented in [FLA10].

**Definition 2.1 (State configuration).** A state configuration  $\mathcal{F}$  consists of:

- A simple graph  $\mathcal{G}$ , i.e., a set  $\text{nodes}(\mathcal{F})$  and a set  $\text{edges}(\mathcal{F})$  where every edge is associated with a 2-element set (unordered pair)  $n \leftrightarrow m$  of nodes. We take  $\text{nodes}(\mathcal{F}) \subseteq \text{COMP}$  (i.e., nodes are components) and  $\text{edges}(\mathcal{F}) \subseteq \text{WIRE}$  (i.e., edges are wires).
- A (configuration) state  $\mathcal{S}$ , which assigns a state  $\mathcal{S}(n) \in \text{STATE}_n$  to every node  $n$  and a state  $\mathcal{S}(e) \in \text{STATE}_e$  to every edge  $e$ .

Every state configuration  $\langle \mathcal{G}, \mathcal{S} \rangle$  can change because either the state function  $\mathcal{S}$  or the graph  $\mathcal{G}$  changes (or both, in the case of components or wires that are not effected by the reconfiguration). Changes to the state

function result from computations executed by components and the coordination activities performed by the wires that connect them. However, the essence of SOC, as we see it in this paper, is not captured at the level of state changes (which is basically a distributed view of computation), but at the level of the changes that operate on configuration graphs: in SOC, changes to the underlying graph of components and wires occur at run time when a component performs an action that triggers the discovery and binding of a service.

Another important aspect of our model is the fact that we view SOC as providing a layer that interacts with two other layers. This can be noticed in Figure 2 where shadows are used for indicating that certain components reside in different layers: LUI and AUI (two user interfaces) belong to what we call the top (or user) layer; DB (a database of users) lies in the bottom (or persistent) layer; all other components execute in the middle (or service) layer.

We use layers as abstractions of organisation and change, not as hierarchies of implementation as found in software engineering where one layer provides the implementation of the design primitives used in the layer above. Our notion of layer is closer to that of 3-tiered architectures, but our layers do not impose a strict hierarchical design methodology in the sense that components are not designed and implemented to sit on a particular layer. Rather, our layers reflect the dynamics of configurations from the point of view of the activities that drive a given business system. This is why there is no formal, static semantics of layers: a configuration is a (flat) graph as indicated above but, in order to understand and model the way it evolves, it is useful to distinguish different levels of dynamicity.

More precisely, in the context of a business activity (for example, an activity executing the request of a client for booking a holiday), the components that execute in the middle layer reflect the sessions of a number of services whose discovery the activity has triggered (e.g., the workflow that orchestrates the travel reservation of that client). These components are created when the session of the corresponding service starts, i.e., as fresh instances that last only for the duration of the session.

The bottom layer consists of the components and connectors that are persistent as far as the service layer is concerned; that is, when a new session of a service starts (for example, a travel agent starts booking a trip on behalf of the activity's client), the components of the bottom layer should be available so that, as the service executes, they can be used as (shared) servers — for example, a database of clients shared by all sessions of the travel booking service, or the reservation log of a hotel, or a currency converter. In particular, the bottom layer can be used for making persistent the effects of services as they execute. In component-based development (CBD) one often says that the bottom layer provides 'services' to the layer above. As we see it in this paper, the difference between CBD and SOC is precisely in the way such services are procured, which in the case of SOC involves identifying (possibly new) providers and negotiating terms and conditions *for each new instance* of the activity, e.g., for each new user of a holiday-booking agent. SOA middleware supports this service layer by providing the infrastructure for the discovery and negotiation processes to be executed without having to be explicitly programmed as (part of) components.

The top layer is the one responsible for launching business activities. More precisely, the user of a given activity resides in the top layer; it can be an interface for human-computer interaction, a software component, or an external system (e.g., a control device equipped with sensors). When the user launches an activity, a component is created in the service layer that starts executing a workflow that may involve the orchestration of services that will be discovered and bound to the workflow at run time. This is explained in more detail in Section 4.2.

### 2.3. Business activities and configurations

In our model, state configurations change as a result of the execution of business processes. More precisely, changes to the configuration graph result from the fact that the discovery of a service is triggered and, as a consequence, new components are added and bound to existing ones (and, possibly, other components and wires disappear because they finished executing). The information about the triggers and the constraints that apply to service discovery and binding are not coded in the components themselves: they are properties of the business activities that are active and determine how the configuration evolves. Thus, in order to formalise the dynamic aspects of SOC, we need to look beyond the information available in a state configuration — state configurations account only for *which* components are active and *how* they are interconnected, not *why* they are active and interconnected in that way.

Business activities are autonomous and active computational ensembles of components that collectively





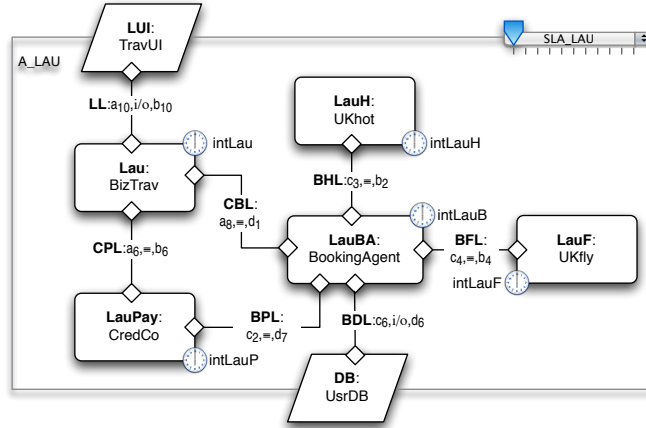




Fig. 4. An activity module

relevant for the purposes of this paper, so we will simply assume that we have available a set  $LAYP$  of specifications of layer protocols and a typing relation on  $COMP \times LAYP$ .

The edges of an activity module are called ‘wire-interfaces’ and labelled with connectors [AG98]. For instance, in Figure 4, the wire-interface  $BHL$  is labelled with the connector  $\langle c_3, \equiv, b_2 \rangle$ . As defined in Section 3.3, a connector consists of an ‘interaction glue’ that describes a protocol between two ‘roles’ (by  $\equiv$  we denote a direct transmission of events [ABFL07]) and two attachments ( $c_3$  and  $b_2$  in the example at hand) linking the roles to the two nodes ( $LauBA$  and  $LauH$ , respectively). In general, the interaction glue may include the routing of events, encryption/decryption of messages, or transforming sent data to the format expected by the receiver. See [ABFL07] for a detailed account of how connectors are formalised in SRML, and [FS07] for a language-independent algebraic semantics. We use  $CNCT$  to designate the set of connectors and  $IGLU$  the set of specifications of the corresponding interaction glue. We also rely on a typing relation  $WIRE \times CNCT$  between wires and connectors. In software architecture in general, connectors may involve an arbitrary number of roles, but service-oriented architectures involve only interactions between two parties.

Finally, an activity module identifies three important aspects related to the way the configuration can evolve and the activity can reconfigure its workflow:

- The external services that the activity may still need to discover and bind to in order to fulfil its business goal.
- An internal configuration policy (indicated by the symbol ) , which identifies the triggers of the external service discovery process as well as the initialisation and termination conditions of the components.
- An external configuration policy (indicated by the symbol ) , which consists of the variables and constraints that determine the quality profile of the activity to which the discovered services need to adhere.

The configuration policies (both internal and external) are discussed in more detail in Section 3.2.

Concerning the external services, activity modules can have multiple (or none) ‘requires-interfaces’, which are labelled by ‘business protocols’ specifying the properties required of the services that need to be procured externally. Requires-interfaces are presented by convex pentagons and placed at the right-hand boundary. For instance, the node  $BA$  labelled by the business protocol  $HotelAgent$  in the activity module depicted in Figure 5 is a requires-interface for a hotel booking service. In SRML, business protocols are specifications written in a temporal logic of stateful interactions [FLA10]. Once again, the exact logic that is used for specifying requires-interfaces is not relevant for this paper, so we assume that we have available a set  $BUSP$  of specifications of business protocols.

Notice that the wire-interfaces that connect the requires-interfaces to the other parties —  $BH$  in the example in Figure 5 — should also be seen as part of the services that are required by the activity. More precisely, they specify the protocols through which the activity wants to interact with the external services.

**Definition 2.2 (Activity module).** An activity module  $M$  consists of:

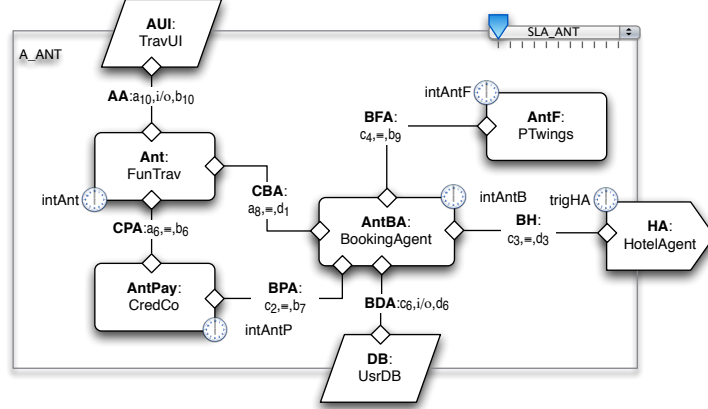


Fig. 5. An activity module with a requires-interface

- A graph  $graph(M)$ .
- A subset of the nodes  $requires(M) \subseteq nodes(M)$ .
- A subset of the nodes  $uses(M) \subseteq nodes(M)$  disjoint from  $requires(M)$ .
- A node  $serves(M) \in nodes(M)$  not belonging to  $requires(M)$  or  $uses(M)$ .
- A labelling function  $label_M$  such that
  - $label_M(n) \in \text{BROL}$  if  $n \in components(M)$ , where  $components(M) = nodes(M) \setminus (requires(M) \cup uses(M) \cup \{serves(M)\})$
  - $label_M(n) \in \text{BUSP}$  if  $n \in requires(M)$
  - $label_M(n) \in \text{LAYP}$  if  $n \in uses(M)$
  - $label_M(serves(M)) \in \text{LAYP}$
  - $label_M(e) \in \text{CNCT}$  if  $e \in edges(M)$
- An internal configuration policy (discussed in Section 3.2).
- An external configuration policy (discussed in Section 3.2).

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

We can now formalise the typing of state configurations with activity modules that we discussed in relation to Figure 3, which accounts for the coarser business dimension that is overlaid by services on global computers. That is, we define what corresponds to a state configuration of a service overlay computer, which we call a business configuration. We consider a space  $\mathcal{A}$  of business activities to be given, which can be seen to consist of reference numbers (or some other kind of identifier) such as the ones that organisations automatically assign when a service request arrives.

**Definition 2.3 (Business configuration).** A business configuration consists of:

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

A homomorphism of graphs is just a mapping of nodes to nodes and edges to edges that preserves the end-points of the edges. Therefore, the homomorphism  $\mathcal{C}$  of a business configuration  $\langle \mathcal{F}, \mathcal{B}, \mathcal{C} \rangle$  types the nodes (components) of  $\mathcal{F}(a)$  with business roles or layer protocols — i.e.,  $\mathcal{C}(a)(n): label_{\mathcal{B}(a)}(n)$  for every node  $n$

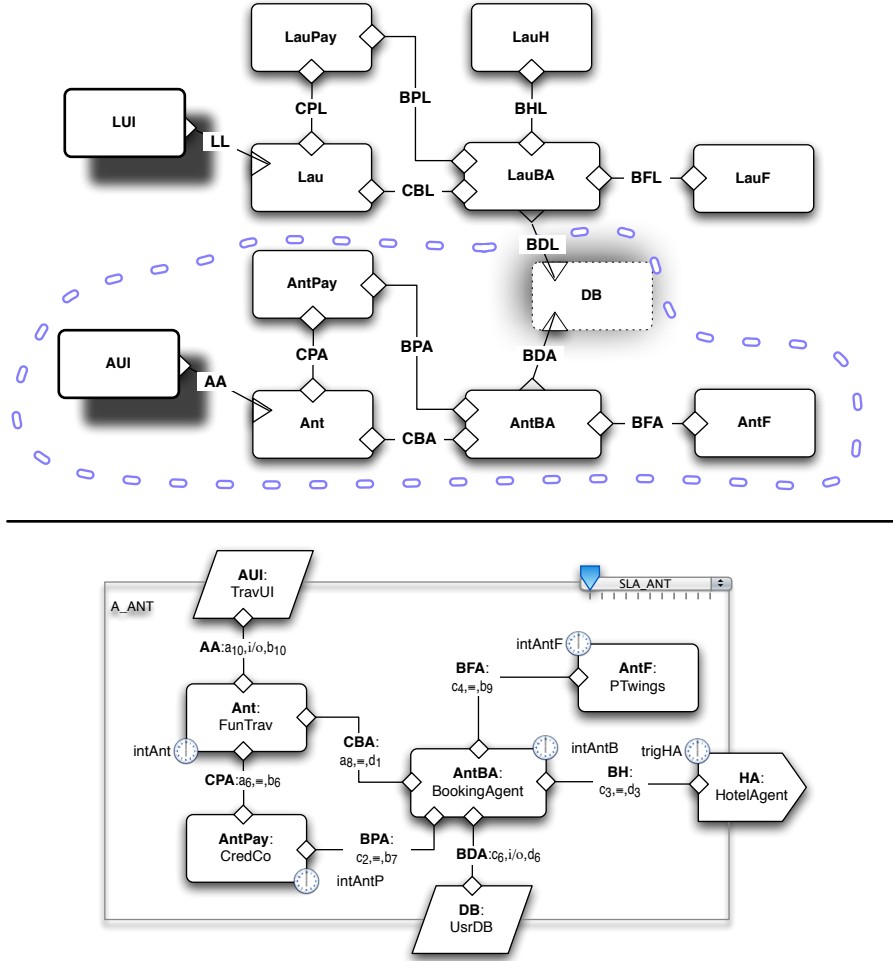


Fig. 6. The sub-configuration and type of the activity ANT executing in a given state configuration

— and the edges (wires) with connectors — i.e.,  $\mathcal{C}(a)(e): label_{\mathcal{B}(a)}(e)$  for every edge  $e$  of the body of the activity. In other words, the homomorphism binds the components and wires of the state configuration to the business elements (interfaces labelled with business roles, layer protocols and connectors) that they fulfil in the activity.

We normally represent the homomorphism  $\mathcal{C}(a)$  by drawing a dashed line around  $\mathcal{F}(a)$  and replacing the nodes of  $\mathcal{B}(a)$  by those of  $\mathcal{F}(a)$  as in Figure 6 for an activity that we will call ANT — the state graph is the one in Figure 2, the sub-configuration  $\mathcal{F}(ANT)$  is the one that we marked in the lower part of Figure 3, and the activity module  $\mathcal{B}(ANT)$  is as in Figure 5. Notice that although we use the same icons for state configurations as for modules, the nodes of modules are not components and the edges are not wires: modules involve abstract models, not instances. The homomorphism establishes the correspondence between the two levels.

The fact that the homomorphism is defined over the body of the activity module means that business protocols are not used for typing components of the state configuration — one can note in Figure 6 that the node HA of the activity module is not mapped to the state configuration. Indeed, as discussed above, the purpose of the requires-interfaces is for identifying dependencies that the activity has, in that state, on external services. In particular, this makes requires-interfaces different from uses-interfaces as the latter are indeed mapped through the homomorphism to a component of the state configuration. The operational semantics that we propose in Section 4.2 for discovery and binding shows that these two kinds of interfaces fulfil totally different purposes: the former are for ‘horizontal’ dynamic composition through service discovery

and binding; the latter are for ‘vertical’ composition with the bottom layer through a ‘classic’ static binding process.

In a sense, the homomorphism makes state configurations *reflective* in the sense of [CBG<sup>+</sup>08] as it adds meta (business) information to the state configuration. This information is used for deciding how the configuration will evolve (namely, how it will react to events that trigger the discovery process). Indeed, reflection has been advocated as a means of making systems adaptable through reconfiguration, which is similar to the mechanisms through which activities evolve in our model.

### 3. Modelling Services

#### 3.1. Service modules

In our approach, services are modelled through ‘service modules’, which are like the activity modules that we discussed in the previous section except that, instead of a serves-interface to the upper layer, they include a ‘provides-interface’ through which activities can connect to the service (identified through a requires-interface). Such interfaces are labelled by business protocols that describe the properties that a customer can expect from the interactions with the service. Uses-interfaces and requires-interfaces can be included in service modules in the same way as in activity modules.

**Definition 3.1 (Service module).** A service module  $M$  consist of:

- A graph  $graph(M)$ .
- A distinguished subset of nodes  $requires(M) \subseteq nodes(M)$ .
- A distinguished subset of nodes  $uses(M) \subseteq nodes(M)$ .
- A node  $provides(M) \in nodes(M)$  not belonging to  $requires(M)$  or  $uses(M)$ .
- A labelling function  $label_M$  such that
  - $label_M(n) \in \text{BROL}$  if  $n \in components(M)$ , where  $components(M) = nodes(M) \setminus (requires(M) \cup uses(M) \cup \{provides(M)\})$
  - $label_M(n) \in \text{BUSP}$  if  $n \in requires(M)$
  - $label_M(n) \in \text{LAYP}$  if  $n \in uses(M)$
  - $label_M(provides(M)) \in \text{BUSP}$
  - $label_M(e: n \leftrightarrow m) \in \text{CNCT}$
- An internal configuration policy (discussed in Section 3.2).
- An external configuration policy (discussed in Section 3.2).

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

In Figure 7 we present the structure of a module that defines a service provided through an interface CR of type **Customer** for booking a flight and a hotel for a given itinerary and dates. The service relies on a component BA of type **BookingAgent** that orchestrates interactions with a service FA of type **FlightAgent** (for booking flights), a service HA of type **HotelAgent** (for booking hotel rooms), a service PA of type **PayAgent** (for handling payments), and an external component DB of type **UsrDB** (that stores information about registered users). The module also makes explicit the connectors that, through the wires, coordinate the communication between the parties involved in the provision of the service. Notice that the wires that connect the provides-interface — CP and CB — are also part of the service offered by the module, i.e., they declare the protocols through which the service can interact with its customers.

The definition of (service) module was inspired by concepts proposed in the Service Component Architecture (SCA) [OSO05]: “SCA is based on the idea that business function is provided as a series of services, which are assembled together to create solutions that serve a particular business need. These composite applications can contain both new services created specifically for the application and also business function from existing systems and applications, reused as part of the composition. SCA provides a model both for the composition of services and for the creation of service components, including the reuse of existing application function within SCA compositions”.

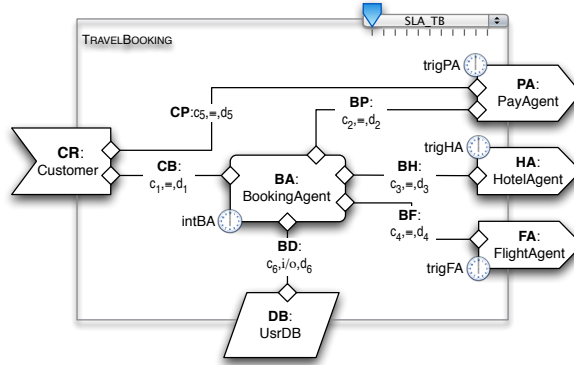


Fig. 7. The structure of a module defining the booking service of a travel agency

Service modules as defined in this paper provide formal abstractions of composite services in the sense of SCA and the way their execution involves a number of external parties that derive from the logic of the business domain. These external parties are not explicitly identified in the module but only implicitly through what we have called requires- and uses-interfaces. As already explained, such interfaces are more than syntactic declarations: they are typed by business protocols — abstract specifications of the conversations in which the parties are required to be involved — or by layer protocols in the case of uses-interfaces — abstract specifications of the interactions supported with the external party. Likewise, the components themselves are not explicitly identified in the module. Instead, the module includes semantic interfaces — business roles — that model the way interactions are orchestrated by the components.

In the case of the provides-interface, the corresponding party is the customer to which the module will be bound to provide a service. The behavioural properties offered in the business protocol that corresponds to the provides-interface result from:

- The tasks performed by the (internal) components that instantiate the business roles;
- The interactions with the external parties that instantiate the requires- and uses-interfaces (and satisfy the properties specified in the corresponding business and layer protocols).

For instance, the business protocol **Customer** could describe that committing to an offer made by the **BookingAgent** ensures that the outcome of the payment process will be acknowledged and that the period during which a booking can be revoked (negotiated prior to binding) is within certain bounds. One of the advantages of working in a formal framework such as ours is that we are able to check that the properties being offered in the provides-interface do result from the orchestration, which in the case of SRML is achieved through model-checking techniques [AMFG09, tBFGM08]. This notion of correctness is defined in Section 3.3.

By ‘customer’ we mean the business activity that triggers the discovery of the service, not the top layer user. Indeed, service modules do not include the serves-interface because, in our model, interactions with the top layer are performed exclusively by business activities. Hence, a user cannot invoke directly a service; a user can launch an activity that, as part of its workflow, may trigger the discovery of a service. As already mentioned, this is, for us, a distinguishing aspect of SOC when compared to CBD. For instance, in the case of the business configuration that corresponds to Figure 3, we can see components — **Lau** and **Ant** — that interact with the users of the activities through two interfaces (**LUI** and **AUI**, respectively). As discussed in Section 4.2, these two interface components are created when the business activities are launched by the corresponding users, orchestrating the interactions with the service components and the corresponding user.

We can use this example to illustrate how our operational semantics of service discovery and binding works. Consider that, in the current state configuration, the top layer entity **AUI** launches a business activity **ANT** typed by the activity module **A\_ANT0** as in Figure 8. (We will not discuss the operational semantics of the top layer, i.e., the mechanism through which entities of the top layer create new components in the service layer.) Consider further that, at a certain point during the execution of the component **Ant**, the trigger condition **trigTA** becomes true and that the service **TravelBooking** as defined in Figure 7 is selected.

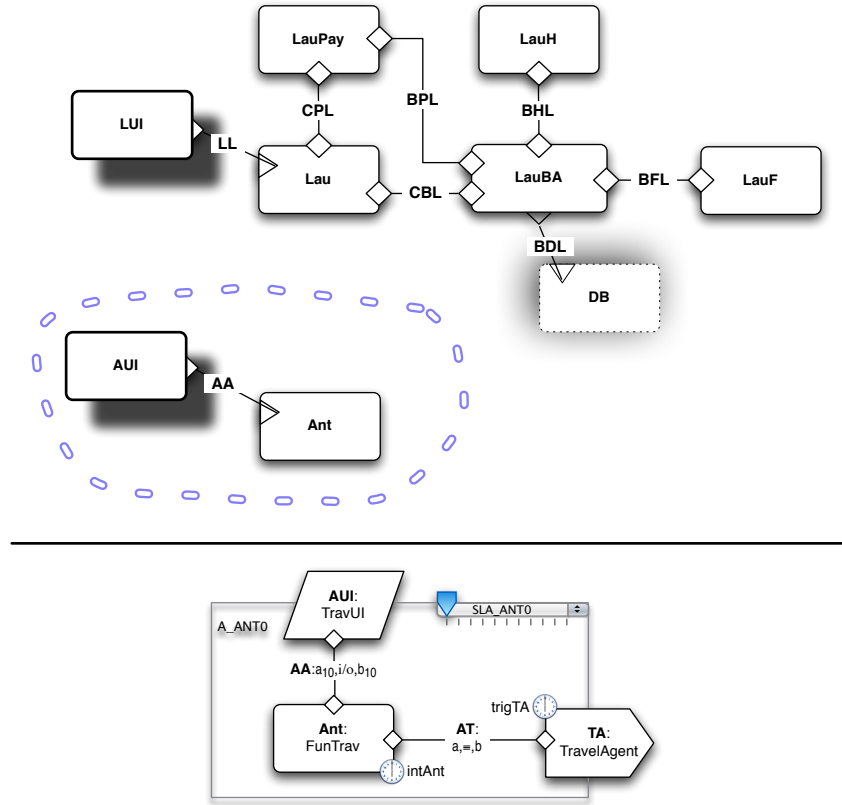


Fig. 8. AUI adds the component Ant to the service layer, typed with FunTrav

As explained in Section 4, this means that the requires-interface TA of A\_ANTO and wire AT can be bound to the provides-interface CR and wires CP and CB of TravelBooking.

The result of the binding is depicted in Figure 9 (and explained in Section 4.2): a new session of TravelBooking starts, which adds the component AntBA to the service layer of the state configuration and connects it to Ant and the component DB of the bottom layer. That is, the workflow of activity ANT is reconfigured and is now typed by the activity module A\_ANT1. Notice that every new session of a service adds to the configuration new instances (components) of its business roles but uses the components already available in the bottom layer.

The configuration depicted in Figure 2 with ANT typed as in Figure 5 would be reached after trigFA and trigPA become true. Notice that, besides the workflow, the external configuration policy also changes as new services are discovered and bound. This process, which determines the ranking and selection of the service, is discussed in Section 4.

### 3.2. The configuration policies

Whereas business roles, business protocols, layer protocols and interaction protocols deal with functional aspects of the behaviour of a (complex) service or activity, configuration policies address properties of the configuration process itself. This is why we focus on them in more detail in this paper.

The internal configuration policy of a module  $M$  concerns the timing of the binding of its interfaces and instantiation of its component and wire interfaces.

**Definition 3.2 (Internal configuration policy).** Given a module  $M$ , its internal configuration policy consists of:

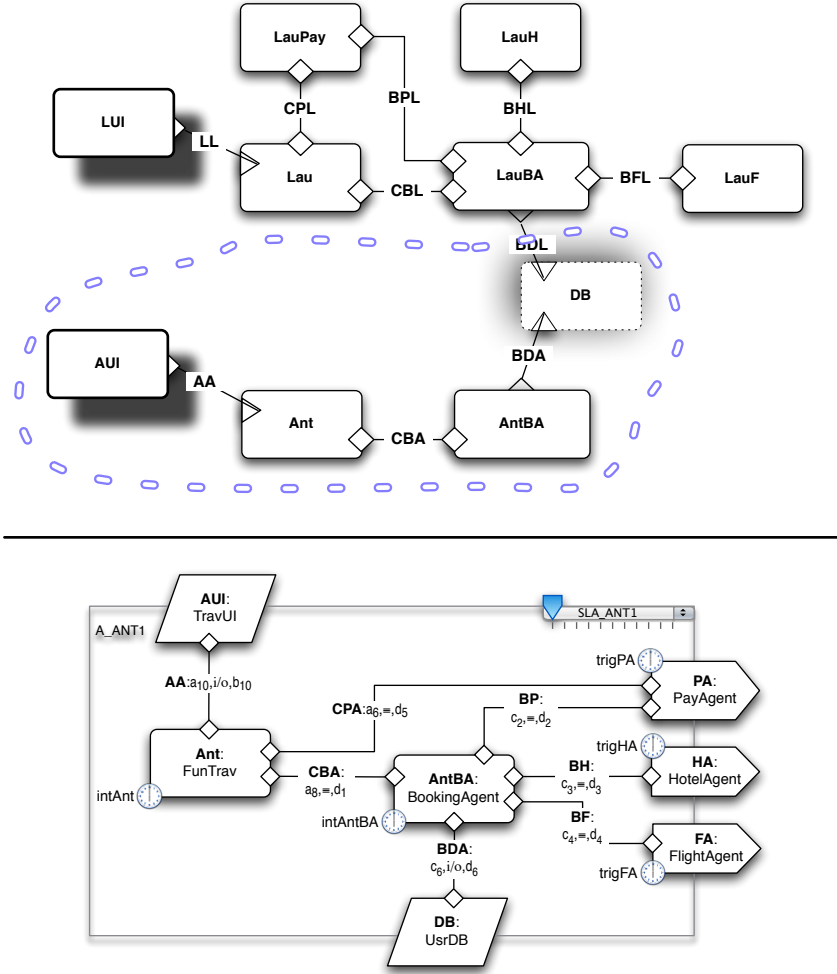


Fig. 9. A new session of TravelBooking starts and reconfigures the workflow of ANT

- For each requires-node  $n \in requires(M)$ , an associated trigger condition  $trigger(n)$ : this is a condition that is evaluated over the state of the activity (i.e., the sub state configuration that corresponds to the activity). When this condition becomes true as a result of a computation step, the process of discovery, selection and binding starts executing, leading to a reconfiguration step (as discussed in Section 4.2) that completes the transition of state configurations. The next computation step takes place in the new configuration, i.e., computations resume when the components of the selected service are instantiated and connected to those of the activity.
- For each component-node  $n \in components(M)$ , an initialisation condition  $init(n)$  that is ensured when the component is instantiated (as discussed in Section 4.2). Typically, these are conditions on the way the state variables of the component are initialised, but they can also include the publication of given events.
- For each component-node  $n \in components(M)$ , a second state condition  $term(n)$  that determines when the component stops executing and interacting with the rest of the components of the activity. Typically, this condition triggers a garbage collection process that removes the associated wires from the configuration, which may also require the delivery of any events pending in the wires (unless we choose to allow for the loss of events).

The external policy concerns the way the module relates to external parties: it declares a set of constraints that have to be taken into account during discovery and selection. Every constraint involves a set of variables

that includes both local parameters of the service being provided (e.g., the percentage of the cost of a trip that is refundable) and standard configuration parameters selected from a fixed set of types — availability, response time, message reliability, *inter alia*. These standard configuration parameters may apply to the service being provided, or to the services that need to be procured externally, or to the wires.

In our approach, we adopt the framework for constraint satisfaction and optimization presented in [BMR97], in which constraint systems are defined in terms of *c*-semirings. As explained therein, this framework is quite general and allows us to work with constraints of different kinds — both hard and ‘soft’, the latter in many grades (fuzzy, weighted, and so on).

**Definition 3.3 (c-semiring).** A *c*-semiring is a semiring  $\langle A, +, \times, 0, 1 \rangle$  in which  $A$  represents a space of degrees of satisfaction, e.g., the set  $\{0, 1\}$  for *yes/no* or the interval  $[0, 1]$  for intermediate degrees of satisfaction. The operations  $\times$  and  $+$  are used for composition and choice, respectively. Composition is commutative, choice is idempotent and 1 is an absorbing element (i.e., there is no better choice than 1). That is, a *c*-semiring is an algebra of degrees of satisfaction. Notice that every *c*-semiring  $S$  induces a partial order  $\leq_S$  (of satisfaction) over  $A$  —  $a \leq_S b$  iff  $a + b = b$ . That is,  $b$  is better than  $a$  iff the choice between  $a$  and  $b$  is  $b$ .

**Definition 3.4 (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).

**Definition 3.5 (Constraint).** A constraint over a constraint system  $\langle S, D, V \rangle$  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.

**Definition 3.6 (External configuration policy).** The external configuration policy of a module  $M$  consists of:

- A constraint system  $cs(M)$  based on a fixed *c*-semiring.
- A set  $sla(M)$  of constraints over  $cs(M)$ .
- For every variable in  $cs(M)$ , a type.
- A partial assignment *owner* of either a node or an edge of  $M$  to each variable of  $cs(M)$ .

For instance, in the case of **TravelBooking**, one could consider the following configuration variables:

- **KD** and **PERC** — parameters of **CR** (customer) that denote the period before departure during which a cancellation is accepted and the percentage of the cost that is refundable in case of cancellation, respectively.
- **RM1** and **RM2** — two different reliable message mechanisms (see [MP04] for examples) that apply to the wires **CB**, **CP** and **BP** (see Figure 7).
- **BOOKFEE** — a parameter of **CR** (customer) and **FA** (flight agent) that denotes the fixed fee for each trip/flight booking transaction, respectively.

Because we are handling 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 [BMR97]. In this *c*-semiring, the preference level is between 0 (worst) and 1 (best). The constraints are:

1.  $\langle \{\text{CR.KD}, \text{CR.PERC}\}, def_1 \rangle$  where

$$def_1(d, p) = \begin{cases} 1 & \text{if } 1 \leq d \text{ and } p \leq 90 \text{ and } p \leq 50 + 50 * d \\ 0 & \text{otherwise} \end{cases}$$

That is, the percentage **PERC** of the cost that is refundable is bounded by the least of 90 percent and a linear function of the period **KD** during which the deal can be revoked (the maximum refundable cost is obtained if the period has 8 or more days).

2.  $\langle \{\text{CB.RM1}, \text{CB.RM2}, \text{CP.RM1}, \text{CP.RM2}, \text{BP.RM1}, \text{BP.RM2}\}, def_2 \rangle$  where

$$def_2(a, b, c, d, e, f) = \begin{cases} 1 & \text{if } a = c = e = 1 \text{ and } b = d = f = 0 \\ 1 & \text{if } a = c = e = 0 \text{ and } b = d = f = 1 \\ 0 & \text{otherwise} \end{cases}$$

That is, the booking agent can use exactly one of the two reliable messaging mechanisms in the conversations with the customer and the pay agent and the same mechanism has to be used with all of them.



3.  $\langle \{FA.BOOKFEE\}, def_3 \rangle$  where

$$def_3(p) = \frac{1}{1+p}$$

That is, the degree of satisfaction for a flight agent is inversely proportional to the booking fee.

4.  $\langle \{CR.BOOKFEE, FA.BOOKFEE\}, def_4 \rangle$  where

$$def_4(d, p) = \begin{cases} 1 & \text{if } p + 5 \leq d \\ 0 & \text{otherwise} \end{cases}$$

That is, the booking fee that the customer has to pay consists of the booking fee of the flight agent plus 5 units for the booking agent service.

As illustrated, we tend to use the dot-notation  $c.X$  (resp.  $w.X$ ) in order to express that component  $c$  (resp. wire  $w$ ) is the owner of the configuration variable  $X$ .

The c-semiring approach also supports selection based on a characterisation of ‘best solution’ supported by multi-dimensional criteria, e.g., minimizing the cost of a resource while maximizing the work it supports. See [BM07] for other usages of this approach for service ranking and selection.

### 3.3. The correctness property of service modules

Service modules are considered to be ‘correct’ when the properties offered in the provides-interface are ensured by the orchestration of its components and the properties specified through its requires-interfaces. For instance, in relation to our running example, we would like to certify that properties offered through the business protocol *Customer* are effectively established by the orchestration performed by *BA* on the assumption that *PA*, *HA* and *FA* are bound to services that deliver the properties required in the corresponding business protocols (*PayAgent*, *HotelAgent* and *FlightAgent*, respectively). An example could be that committing to an offer made by *BA* ensures that the outcome of the payment process will be acknowledged; and another that the period during which a booking can be revoked (negotiated prior to binding) is within certain bounds.

This correctness property of modules is best expressed in terms of logical entailment. The mechanisms that we provide for putting together, interconnecting and composing modules is largely independent of any such logic. The particular choice of logic operators, their semantics and proof-theory are essential for supporting the modelling of service-based applications but not for the semantics and pragmatics of composing modules. For the purpose of this paper, what is important is that the logic satisfies some structural properties that are required for the correctness condition and the notion of module composition to work well together as explained below. Notice that this generic characterisation is also agnostic in relation to the specific techniques through which correctness can be checked. In SRML, we have been investigating the use of model-checking techniques [AMFG09, tBFGM08] in particular.

More specifically, we assume that we have available an entailment system (or  $\pi$ -institution) [Fia04]  $\langle \text{SIGN}, \text{gram}, \vdash \rangle$ . By *SIGN* we denote the category of signatures of the logic, each of which represents a set of interactions; signature morphisms are maps that preserve the structure of interactions (whether they are synchronous or asynchronous, their parameters, and so on). The grammar functor  $\text{gram}: \text{SIGN} \rightarrow \text{SET}$  generates, for each signature, the language used for describing properties of the interactions. Notice that, given a signature morphism  $\sigma: Q \rightarrow Q'$ ,  $\text{gram}(\sigma)$  translates properties in the language of  $Q$  to the language of  $Q'$ .

We denote by  $\vdash_Q$  the entailment system that allows us to reason about properties in the language of  $Q$ . We write  $S \vdash_Q s$  to indicate that sentence  $s$  is entailed by the set of sentences  $S$ . Pairs  $\langle Q, S \rangle$  consisting of a set  $S$  of sentences over a signature  $Q$  — usually called theory presentations — can be organised in a category *SPEC* whose morphisms capture what are usually called ‘interpretations between theories’. We denote by  $\text{sign}$  the functor  $\text{SPEC} \rightarrow \text{SIGN}$  that projects theory presentations to the underlying signatures. We further assume that *SPEC* is finitely co-complete — i.e., that there is a canonical (minimal) way of amalgamating a diagram of specifications — and coordinated over  $\text{sign}$  [Fia04] (implying that we can amalgamate mixed diagrams of specifications and signatures).

Business roles, layer protocols, business protocols and interaction protocols carry a semantic meaning that we take to be defined by mappings:  $\text{spec}_{\text{BROL}}: \text{BROL} \rightarrow \text{SPEC}$ ,  $\text{spec}_{\text{LAYP}}: \text{LAYP} \rightarrow \text{SPEC}$ ,  $\text{spec}_{\text{BUSP}}: \text{BUSP} \rightarrow \text{SPEC}$  and  $\text{spec}_{\text{IGLU}}: \text{IGLU} \rightarrow \text{SPEC}$ , respectively. In the case of business roles, this assumes that we can abstract properties from orchestrations, which corresponds to defining an axiomatic semantics of an orchestration language. In the case of business and interaction protocols, this mapping is more of a translation

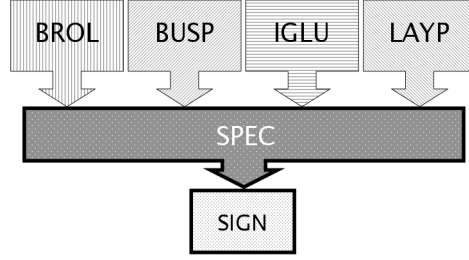


Fig. 10. Relating the specification domain with the other formal domains

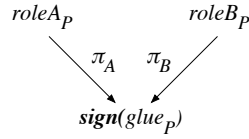


Fig. 11. Interaction protocols as structured co-spans

from the language of interactions to a logic in which one can reason about their properties as well as that of orchestrations. For simplicity, we shall also denote by  $sign_X$  the compositions of the functors  $spec_X$  with  $sign$ , which we often simplify to  $sign$  if there is no risk of confusion. We summarise the relationships between all these formalisms in Figure 10.

To formulate the correctness property of modules using these formalisms, we need to map modules to diagrams in SPEC. In order to define this mapping, we need to be more precise on what we mean by a (binary) connector — an interaction protocol together with attachments to the parties being connected.

**Definition 3.7 (Interaction protocol).** An interaction protocol  $P$  is a triple  $\langle \pi_A, glue_P, \pi_B \rangle$  where

- $glue_P$ : IGLU
- $\pi_A: roleA_P \rightarrow sign(glue_P)$  and  $\pi_B: roleB_P \rightarrow sign(glue_P)$  are signature morphisms.

The specification  $glue_P$  — called the ‘glue’ — describes the coordination mechanisms enforced by the protocol. The signatures  $roleA_P$  and  $roleB_P$  — the ‘roles’ of the protocol — act as formal parameters. The morphisms  $\pi_A$  and  $\pi_B$  identify the interactions involved in the roles of the protocol. This algebraic semantics identifies every interaction protocol with a structured co-span [FS07] as depicted in Figure 11.

**Definition 3.8 (Connector).** A connector is a triple  $\langle \mu_A, P, \mu_B \rangle$  where:

- $P$  is an interaction protocol  $\langle \pi_A, glue_P, \pi_B \rangle$ .
- $\mu_A$  and  $\mu_B$  — the ‘attachments’ — are signature morphisms with sources  $roleA_P$  and  $roleB_P$ , respectively.

Using this categorical view of connectors, we can now expand the labelled graph associated with every module to define a proper diagram. This construction needs to take into account three different kinds of edges: those that connect internal components, the provides-interface, and the requires-interfaces.

Given a module  $M$ , let  $e: n \leftrightarrow m$  be an edge such that neither  $n$  nor  $m$  belongs to  $requires(M)$  and, if  $M$  is a service module, neither  $n$  nor  $m$  is  $provides(M)$ . For  $M$  to be well typed, it is necessary that the attachments of the connector  $\langle \mu_A, P, \mu_B \rangle$  that labels  $e$  are morphisms  $\mu_A: roleA_P \rightarrow sign(label_M(n))$  and  $\mu_B: roleB_P \rightarrow sign(label_M(m))$ . As shown in Figure 12, such a connector defines a diagram in SIGN.

In the case of a service module  $M$ , the connectors that label the wires  $e_i: provides(M) \leftrightarrow m_i$  need to be treated in a special way because the ‘customer’ is not yet available and, therefore, the corresponding attachments cannot be defined. This is why we choose a default signature that consists of all the interactions that are involved in the protocols. More precisely, if the corresponding connectors  $label_M(e_i)$  are of the form  $\langle \mu_i, P_i, \mu_{m_i} \rangle$ , the signature that represents the (logical) customer is the sum  $\bigoplus_j roleA_{P_j}$  of all the roles, and the attachments are the inclusions  $roleA_{P_i} \rightarrow \bigoplus_j roleA_{P_j}$ . The reason we take the sum is that this is the biggest set of interactions that the corresponding protocol coordinates. Therefore, for a module to be well

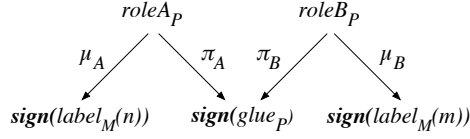


Fig. 12. The diagram defined by a well-typed connector

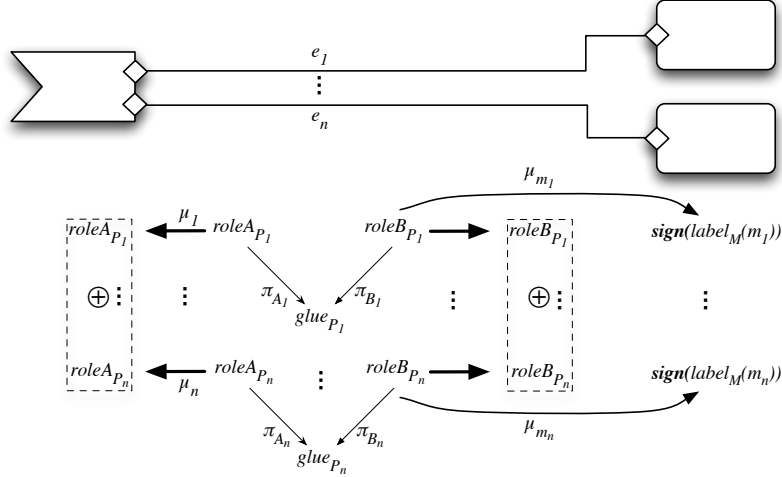


Fig. 13. The types of the wires connecting the provides-interface

typed, we also need that all the attachments to  $provides(M)$  be inclusions  $roleA_{P_i} \rightarrow \oplus_j roleA_{P_j}$  as depicted in Figure 13.

On the other hand, although we defined the label of  $provides(M)$  to be a business protocol, that specification represents the properties of the service offered by the module, not the customer with which the wires  $e_i$  should be connected. Therefore, the signature of  $label_M(provides(M))$  should be the sum of the other set of roles  $\oplus_j roleB_{P_j}$ , i.e., the properties of the provided service should be written in the language of this signature and not that of the customer. In summary, the signature of the business protocol that labels the provides-interface is not the signature to which the wires are attached (which is that of a logical customer) but the sum of the interactions supported by the entities connected to it.

A similar condition applies to the requires-interfaces. More precisely, the signature of the business protocols that label the requires-interface should not require more interactions than the ones involved in the wires that connect them to the body of the module (be it an activity or a service module). Therefore, the signature of a business protocol should be a (disjoint) sum of the roles  $\oplus_j roleB_{P_j}$  of the interaction protocols involved in the connections as depicted in Figure 14.

**Definition 3.9 (Well-typed module).** We say that a module  $M$  is well typed iff:

- If  $M$  is a service module, let  $e_i: provides(M) \leftrightarrow m_i$  be all the edges (wires) that connect the node  $provides(M)$  and  $\langle \mu_i, P_i, \mu_{m_i} \rangle$  the corresponding labels (connectors). The signature of the business protocol  $label_M(provides(M))$  is (isomorphic to) the sum  $\oplus_j roleB_{P_j}$  and  $\mu_i: roleA_{P_i} \rightarrow \oplus_j roleA_{P_j}$  for every attachment.
- For every  $R \in requires(M)$ , let  $e_i: m_i \leftrightarrow R$  be all the edges (wires) that connect  $R$  and  $\langle \mu_i, P_i, \mu_{m_i} \rangle$  the corresponding labels (connectors). The signature of the business protocol  $label_M(R)$  is (isomorphic to) the sum  $\oplus_j roleB_{P_j}$  and we have  $\mu_{m_i}: roleA_{P_i} \rightarrow \oplus_j roleA_{P_j}$  for every attachment.
- For every other edge  $e: n \leftrightarrow m$  let  $\langle \mu_A, P, \mu_B \rangle$  be its label. In this case, we have  $\mu_A: roleA_P \rightarrow sign(label_M(n))$  and  $\mu_B: roleB_P \rightarrow sign(label_M(m))$ .

**Proposition and Definition 3.1 (expanded(M)).** Given a module  $M$  that is well typed, we construct another labelled graph  $expanded(M)$  as follows:

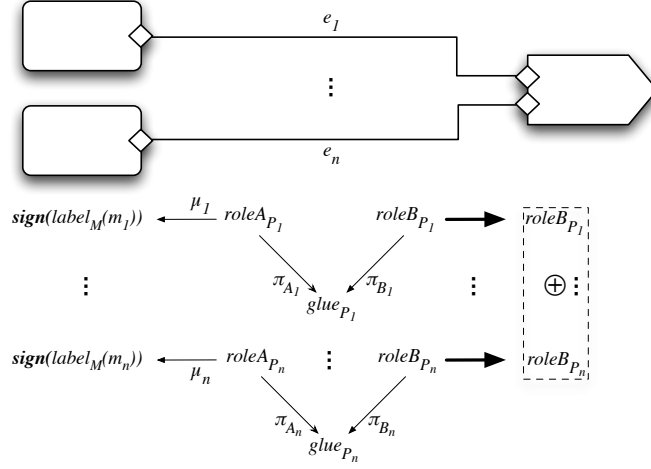


Fig. 14. The types of the wires connecting a requires-interface

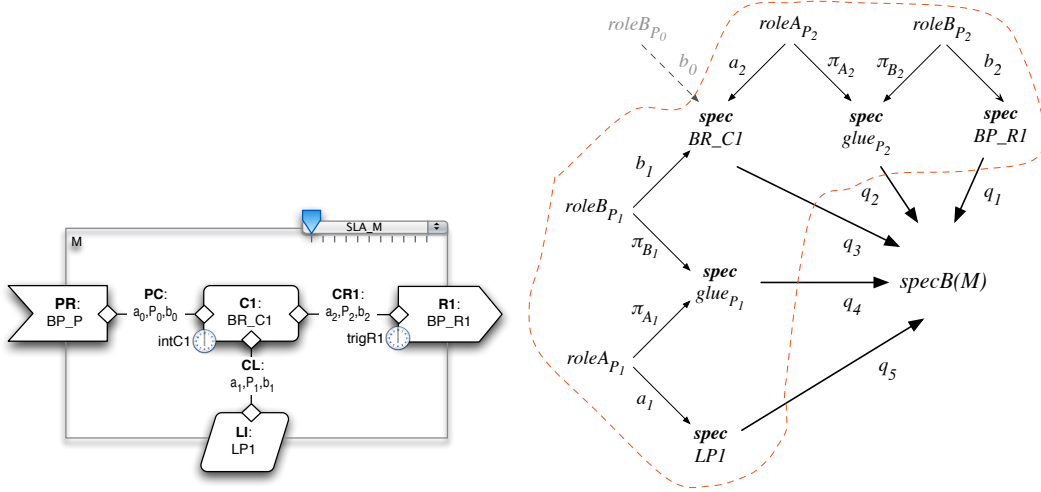


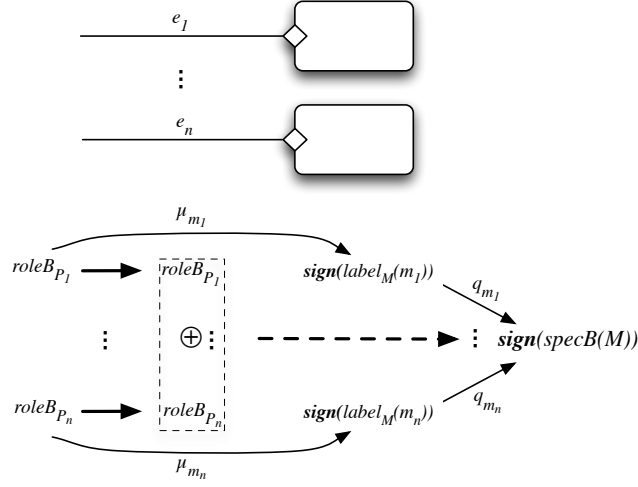
Fig. 15. Example of  $expanded(M)$  — inside the dashed line — and its colimit. We also indicate (in grey) the morphism that connects the signature of the provides-interface to the expanded diagram

- We remove the provides-interface and all the edges that connect it.
- For every edge (wire) of  $M$  that remains, we replace it by the diagram defined by its connector as in Figure 12, which adds three new nodes (for the glue and roles) and four directed edges.

An example is given in Figure 15.

**Proposition and Definition 3.2** ( $specA(M)$ ). Given a module  $M$  that is well typed, if we apply the mapping  $spec$  to the business roles (of the component interfaces), the layer protocols (of the serves- and uses-interfaces), the glues of the interaction protocols, and the business protocols (of the requires-interfaces) of  $expanded(M)$ , we obtain a diagram in the (coordinated) category  $SPEC$ . The colimit (amalgamated sum) of this diagram returns a specification  $specA(M)$  and, for every node  $n$  of  $expanded(M)$ , a morphism  $q_n: sign(label_M(n)) \rightarrow sign(specA(M))$ .

**Definition 3.10** ( $specB(M)$ ). Given a module  $M$  that is well typed, we define the specification  $specB(M)$  that results from adding to  $specA(M)$  the properties that derive from the internal configuration policy, which include the initialisation and termination constraints as well as the triggers associated with the requires-interfaces of  $M$ .



**Fig. 16.** How the signature of the business protocol of the provides-interface maps to  $sign(specB(M))$  for a generic  $M$

As for the business protocol of  $provides(M)$ , we have already defined its signature as being the sum  $\oplus_j roleB_{P_j}$  of the roles of the connectors that label the wires  $e_i : provides(M) \leftrightarrow m_i$ .

**Proposition and Definition 3.3** ( $specP(M)$ ). The attachments  $\mu_{m_i} : roleB_{P_i} \rightarrow sign(label_M(m_i))$  composed with the morphisms  $q_{m_i} : sign(label_M(m_i)) \rightarrow sign(specB(M))$  extend to a morphism  $\oplus_j roleB_{P_j} \rightarrow sign(specB(M))$  (see Figure 13 and Figure 16). We denote by  $specP(M)$  the translation of the specification of the business protocol of  $provides(M)$  induced by that morphism.

*Proof.* The result follows from the universal properties of the sum.  $\square$

Given that all these sets of sentences are now in the language of  $specB(M)$ , we can define a correctness property for service modules.

**Definition 3.11 (Correctness).** We say that a service module  $M$  is correct iff  $specB(M) \vdash specP(M)$ .

That is, a module is correct iff the properties of the provides-interfaces are entailed by the module assuming that the properties of the requires-interfaces and their connectors hold.

Naturally, another important property is *consistency*, i.e. that  $specB(M)$  is free from contradictions (or, equivalently, that it admits a model). Given that the specifications involved in the colimit are generated by the mappings  $spec$  applied to business roles, business protocols, layer protocols and interaction protocols, ensuring their consistency should not be difficult. Consistency of the whole module depends, essentially, on the nature of the connections established by the wires. To state the obvious, specific strategies for ensuring consistency will depend on the nature of the particular formalisms involved. Examples of formal techniques that can be used for this purpose are [BBC<sup>+</sup>06, CHY07, VCS08]. See also [FLA10] for a discussion on consistency for the specific computational and coordination models developed for SRML.

## 4. The Operational Semantics of Service Discovery and Binding

### 4.1. Unification: discovery, ranking and selection

As mentioned in Section 3.2, every module declares, as part of its internal configuration policy, a triggering condition for each requires-interface. This is the condition that determines when a service needs to be discovered and bound to the current configuration through that interface. More precisely, given a business configuration  $\mathcal{BC} = \langle \langle \mathcal{G}, \mathcal{S} \rangle, \mathcal{B}, \mathcal{C} \rangle$  and an activity  $a$ , each condition  $trigger(R)$  where  $R \in requires(\mathcal{B}(a))$  is evaluated over the state  $\mathcal{S}$ . If the condition  $trigger(R)$  for a given requires-interface  $R$  holds in  $\mathcal{BC}$ , the ‘unification’ process is launched, which should return a service that ‘best’ fits the business protocol

$label_{\mathcal{B}(a)}(R)$ , the interaction protocols of the wires that connect the requires-interface to the rest of the module, and the external configuration policy of  $\mathcal{B}(a)$ .

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

- *Discovery.* This step consists in finding the services — among those that are able to guarantee the properties of the business protocol  $label_{\mathcal{B}(a)}(R)$  associated with  $R$  and of the interaction protocols that label the wires that connect  $R$  — with which it is possible to reach a service-level agreement.
- *Ranking.* For each service  $M$  discovered in the previous step, we calculate the most favourable service-level agreement that can be achieved — the contract that will be established between the two parties if  $M$  is selected. This calculation uses a notion of satisfaction that takes into account the preferences of the activity  $a$  and the service  $M$ .
- *Selection.* One of the services that maximises the level of satisfaction offered by the corresponding contract is selected.

We are now going to define each of these steps in more detail.

**Proposition and Definition 4.1 (Discovery).** Consider a business configuration  $\mathcal{BC} = \langle \mathcal{F}, \mathcal{B}, \mathcal{C} \rangle$  and let  $R$  be a requires-interface of a business activity  $a$  such that  $trigger(R)$  holds in  $\mathcal{F}$ . The discovery phase returns pairs  $\langle M, \rho \rangle$  where  $M$  is a service module and  $\rho$  is a mapping satisfying the properties below. Let  $PR$  be the provides-interface of  $M$ , i.e.,  $PR = provides(M)$ .

- For every wire  $w: s \leftrightarrow R$  of  $a$ , there is a non-empty set  $\rho(w)$  of wires of  $M$  of the form  $PR \leftrightarrow q$  (the set of wires that unify with  $w$ ). For every wire  $v: PR \leftrightarrow q$  of  $M$  there is  $w: s \leftrightarrow R$  of  $a$  such that  $v \in \rho(w)$ , i.e., each wire on the side of the provides-interface unifies with at least one wire on the requires-interface side.
- Let  $\oplus \rho(w)$  be the sum of the interaction protocols of the wires in  $\rho(w)$  as co-spans, and  $\langle \pi_A, glue_P, \pi_B \rangle$  the interaction protocol of  $w$ . The mapping  $\rho$  defines a morphism  $\langle \pi_A, glue_P, \pi_B \rangle \rightarrow \oplus \rho(w)$  of co-spans, i.e., a morphism in IGLU between  $glue_P$  and the sum of the glues of the interaction protocols of the wires in  $\rho(w)$  and signature morphisms between the roles that commute with the morphism of glues.
- The collection of signature morphisms of the form  $role_{B_P} \rightarrow \oplus_{v \in \rho(w)} role_{B_v}$  for all wires that connect  $R$  extends to a morphism  $label_{\mathcal{B}(a)}(R) \rightarrow label_M(PR)$  of specifications, i.e., the behavioural properties offered by the provides-interface of  $M$  entail the properties required by the requires-interface of the activity up to a suitable translation between the languages of both.
- The constraint system  $cs(M)$  of the external policy of  $M$  is compatible with that of  $cs(\mathcal{B}(a))$ . This means that the mapping  $\rho$  is such that, for every variable  $v \in cs(\mathcal{B}(a))$ :
  - if  $owner(v) = R$ , there exists  $\rho(v)$  in  $cs$  such that  $type(v) = type'(\rho(v))$  and  $owner'(\rho(v)) = PR$ ;
  - if  $owner(v)$  is a wire  $w: s \leftrightarrow R$  then, for every  $w' \in \rho(w)$  there is a variable  $\rho(v, w')$  in  $cs(M)$  s.t.  $owner'(\rho(v, w')) = w'$  and  $type(v) = type'(\rho(v, w'))$ .
- The combination  $sla(\mathcal{B}(a)) \oplus_{R, \rho} sla(M)$  of the sets of constraints of  $\mathcal{B}(a)$  and  $M$  is consistent (as defined below, in Definition 4.1).

Intuitively, compatibility means that each discovered service needs to support the negotiation of the configuration parameters associated with  $R$ , i.e., those configuration parameters that belong to  $R$  or to the wires that connect  $R$  to the components of the activity module. The last condition ensures that it is indeed possible to achieve a service-level agreement between the activity and the service module. Compatibility of the constraint systems of  $\mathcal{B}(a)$  and  $M$  relative to  $R$  ensures that they can be combined, which gives rise to another constraint system.

**Definition 4.1 (Combined constraint system).** The combined constraint system  $cs(\mathcal{B}(a)) \oplus_{R, \rho} cs(M)$  is defined as follows:

- Its domain  $D''$  is the union  $D \cup D'$  of the domains of  $cs(\mathcal{B}(a))$  and  $cs(M)$ .
- Its set of variables  $V$  is the disjoint union of  $cs(\mathcal{B}(a))$  and  $cs(M)$  except that each pair of variables of the form  $\langle v, \rho(v) \rangle$  or  $\langle v, \rho(v, w') \rangle$  gives rise to a single variable denoted  $v|\rho(v)$  or  $v|\rho(v, w')$ , respectively — the variable involved in the negotiation. Notice that, if  $owner(v)$  is a wire  $w: i \leftrightarrow R$ , then we may end up with several ‘aliases’  $v|\rho(v, w')$ , one for each wire  $w' \in \rho(w)$ , i.e., that  $\rho$  pairs with  $w$ . We denote by  $neg(R, \rho)$  the set of such variables.

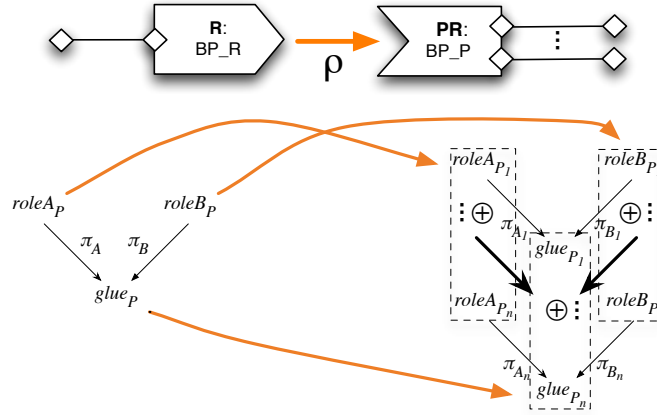


Fig. 17. Unifying wires

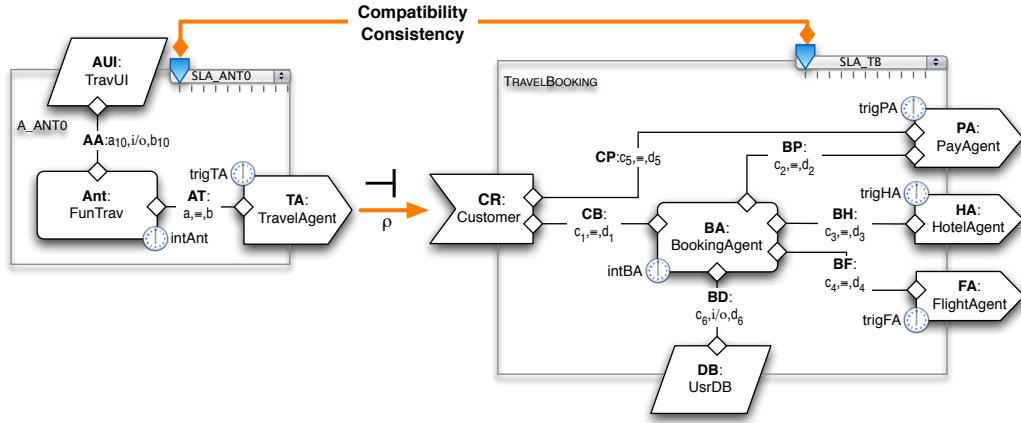


Fig. 18. The elements involved in unification

The combined set of constraints is defined by ‘lifting’ the constraints of  $sla(\mathcal{B}(a))$  and  $sla(M)$  to the new constraint system, for which two aspects need to be accounted for. On the one hand, we have to take into account that variables may have multiple aliases, which requires constraints to be replicated (one for each alias). On the other hand, we have to account for the fact that constraints lifted from any of the modules involve variables from the other module; in such cases, we define that the degree of satisfaction of any tuple assigning new domain elements to old variables is 0. Formally, the lifting operates as follows:

**Definition 4.2 (Combined set of constraints).** The combined set of constraints  $sla(\mathcal{B}(a)) \oplus_{R,\rho} sla(M)$  is defined by:

- Lifting every constraint  $\langle con, def \rangle$  over  $cs(\mathcal{B}(a))$  as follows:
  - If  $con$  does not involve any variable being negotiated (i.e.,  $neg(R,\rho)$ ) we lift the constraint to  $\langle con, def'' \rangle$  where  $def''$  coincides with  $def$  on  $D^{|con|}$  and assigns the value 0 elsewhere.
  - For every  $v \in con$  that belongs to  $neg(R,\rho)$  and alias  $v|v'$ , we lift the constraint to  $\langle con'', def'' \rangle$  where  $con''$  is obtained from  $con$  by replacing each such variable by the chosen alias, and  $def''$  coincides with  $def$  on  $D^{|con|}$  and assigns the value 0 elsewhere.
- Performing the same lifting on the constraints of  $cs(M)$ .

It remains to discuss what we mean by consistency of the combined set of constraints.

**Definition 4.3 (Consistency of a set of constraints).** The consistency of a set of constraints is defined

in terms of the notion best level of consistency, which assigns an element of the  $c$ -semiring to every set of constraints  $C$  as follows (for more details see [BMR97]):

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

Intuitively, this notion gives us the degree of satisfaction that we can expect for the set of constraints of a given problem. A set of constraints  $C$  is said to be consistent iff  $blevel(C) >_S 0$ .

In order to illustrate these constructions (see Figure 18), consider the business activity A\_ANT0 (see also in Figure 8). Consider that, at a certain point of the corresponding workflow, the condition `trigTA` becomes true and triggers the unification process for TA. For the service module `TravelBooking` (see Figure 7) to be discovered, we would need to

- Establish a specification morphism between `TravelAgent` (the business protocol that types TA) and `Customer` (the business protocol that types the provides interface CR of `TravelBooking`) showing that the properties required in `TravelAgent` are entailed by those of `Customer`.
- Check that the constraint systems of A\_ANT0 and `TravelBooking` are compatible.

Entailment is handled through the logic that is adopted for SPEC as discussed in Section 3.3. In order to illustrate what we mean by compatibility, suppose that the constraints imposed by the external policy SLA\_ANT0 involve uniquely the variables TA.PERC, TA.KD, AT.RM1, and require that:

- The reliable messaging mechanism RM1 should be used in the conversations with TA over the wire AT — a hard constraint.
- The degree of satisfaction associated with a refund  $p$  and a period for revoking of duration  $d$  is given by  $p/(d * 100)$  — a soft constraint.

In this situation, it is easy to check compatibility. The constraint system of `TravelBooking` includes the variables CR.PERC, CR.KD, CB.RM1, CP.RM1 and, hence, their values can be negotiated establishing the pairs TA.PERC|CR.PERC, TA.KD|CR.KD, AT.RM1|CB.RM1, AT.RM1|CP.RM1. Notice that AT.RM1 has two aliases because CR is linked by two wires CB and CP. This implies that the constraint (1) over AT.RM1 is lifted to two constraints: one over AT.RM1|CB.RM1 and the other over AT.RM1|CP.RM1. That is, in the combined system, CB and CP have to use the reliable messaging mechanism RM1. Indeed, it is not difficult to conclude that no inconsistencies arise when we combine the two sets of constraints. For instance, the level of satisfaction associated with a situation in which TA.PERC|CR.PERC=55, TA.KD|CR.KD=1, AT.RM1|CB.RM1=1, AT.RM1|CP.RM1=1 would be 0.55; this would be a possible service-level agreement if `TravelBooking` were to be selected to be bound to the business activity A\_ANT0.

Finally, we can discuss how contracts are established.

**Proposition and Definition 4.2 (Ranking).** Together with the set  $neg(R, \rho)$  of the variables being negotiated (those in the domain of  $\rho$ ), the set of constraints  $sla(\mathcal{B}(a)) \oplus_{R, \rho} sla(M)$  defines a constraint problem. The solution of this constraint problem is again a constraint and, hence, it assigns a degree of satisfaction to each possible tuple of values for the variables in  $neg(R, \rho)$ . The outcome of the negotiation between the business activity  $a$  and the provider of  $M$  is any tuple whose associated degree of satisfaction is  $blevel(sla(\mathcal{B}(a)) \oplus_{R, \rho} sla(M))$ , which is again a constraint. That is, a service level agreement is simply an assignment of values to the configuration parameters in  $neg(R, \rho)$ . Hence, ranking a discovered service  $M$  in our framework consists in finding an assignment that maximizes the degree of satisfaction. We denote by  $contract(\mathcal{B}(a) \oplus_{R, \rho} M)$  the constraint that results from the negotiation.

In our example, there are other assignments of values to TA.PERC|CR.PERC and TA.KD|CR.KD that return a positive degree of satisfaction: for instance, 60 and 6 return 0.1. However, the previous assignment maximizes this value, which is why it represents a possible service level agreement between `conf` and the provider of the `TravelBooking` service.

**Definition 4.4 (Selection).** Selection consists in choosing one service from the set of discovered services, taking into account its ranking — the degree of satisfaction assigned to the outcome of the negotiation. The selected service is one with maximal rank.

In our example, the value 0.55 would have to be compared with the outcome of the negotiation of A\_ANT0



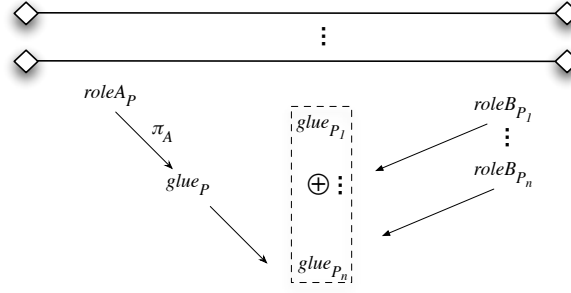


Fig. 19. The wires that result from Figure 17

with other discovered travel agency services. If all the other outcomes were worse than 0.55, TravelBooking would be the selected service.

## 4.2. The reconfiguration step

It remains to define the new business configuration that results from the process of discovery, ranking and selection — what we call the ‘reconfiguration step’. This includes the new state configuration that results from instantiating the selected service over the current configuration and binding it to the business activity  $a$  that triggered the process, and the typing of the business activity with a new module. We start by defining the activity module that will type  $a$  in the new business configuration.

**Proposition and Definition 4.3 (Composition of modules).** Consider a service module  $M$  returned by the selection process upon the occurrence of  $trigger(R)$  where  $R$  is a requires-interface of  $\mathcal{B}(a)$ . The binding of  $R$  with an instance of  $M$  involves the assembly of modules  $\mathcal{B}(a)$  and  $M$ , giving rise to a new module that corresponds to the new execution plan of  $a$ . This new module is the composition  $\mathcal{B}(a) \oplus_{R,\rho} M$  defined as follows:

**The graph and its labelling** The graph of  $\mathcal{B}(a) \oplus_{R,\rho} M$  is obtained from the sum (disjoint union) of the graphs of  $\mathcal{B}(a)$  and  $M$  by eliminating the nodes  $R$  and  $provides(M)$  and the edges that connect them, and adding an edge  $i \leftrightarrow j$  between any two nodes  $i$  and  $j$  such that  $w: i \leftrightarrow R$  is an edge of  $\mathcal{B}(a)$  and  $provides(M) \leftrightarrow j \in \rho(w)$ . The requires-interfaces are those of  $\mathcal{B}(a)$ , except for  $R$ , and those of  $M$ . Given that  $provides(M)$  has been eliminated, there are no provides-interfaces; we obtain an activity module  $B$  that defines the new workflow of the activity  $a$ .

The labels of the resulting graph are inherited from the graphs of  $\mathcal{B}(a)$  and  $M$ , except for the new edges  $i \leftrightarrow j$  that result from the binding of  $R$  and  $provides(M)$  through the mapping  $\rho$ . These are calculated by composing the connectors that label  $w: i \leftrightarrow R$  and each  $provides(M) \leftrightarrow j \in \rho(w)$ . This process of composition works as depicted in Figure 19: basically, we take the glue to be the sum  $\oplus \rho(w)$  and the roles of  $i \leftrightarrow R$  and  $provides(M) \leftrightarrow j$ . This process is repeated for every pair of wires that are connected through the mapping  $\rho$ .

**The external configuration policy** This is given by the combined constraint system  $cs(\mathcal{B}(a)) \oplus_{R,\rho} cs(M)$  and the constraints of the sum  $sla(\mathcal{B}(a)) \oplus_{R,\rho} sla(M)$  together with  $contract(\mathcal{B}(a) \oplus_{R,\rho} M)$  — the result of the negotiation as defined in Section 4.1. Owners and types are inherited except for the pairs of variables involved in the negotiation (which no longer require an owner).

**The internal configuration policy** Triggers, initialisation and termination conditions are all inherited from  $\mathcal{B}(a)$  and  $M$ .

We take this module to provide the reconfigured workflow of the business activity  $a$ . Notice that the composition of the wires involves only the glues of the protocols, not the components. In [Fia04, WLF01], we have shown how such forms of composition can be supported in CommUnity — an architectural description language. See also [BS08] for an algebraic formalisation of connector composition in the Behaviour-Interaction-Priority (BIP) component framework.

We can now define the new state and business configurations that result from the discovery and binding processes.

**Definition 4.5 (The reconfigured state configuration).** The current state configuration is modified as follows:

- New components (nodes) are added (to the service layer), which are typed by the business roles of  $components(M)$ .
- New wires (edges) are added that are typed with the connectors that link together the new components introduced in the previous step.
- New wires are added between the new components and the ones that were already present in the configuration, which are typed by the composed connectors that result from the bindings.
- New wires are added that bind the new service components to components (of the bottom layer) that are typed by the layer protocols of  $uses(M)$ . Notice that we do not create new components (instances) in the bottom layer.
- The new components and wires are chosen such that the negotiated conditions on configuration variables are enforced, and initialised so as to satisfy the internal configuration policy of  $M$ .

The state of components and wires not affected by the reconfiguration can also change during a reconfiguration step as a result of the computations that they execute.

Notice how this operational semantics differentiates between the different kinds of nodes and their specifications that we introduced in Sections 2.3 and 3.1. The difference between business roles and protocols is that components that correspond to the business roles are created and bound to their interfaces when the module is instantiated (i.e., when a new session of the service is initiated) whereas the external services that correspond to the business protocols are bound to the require interfaces at run time after a process of discovery, ranking and selection triggered according to the internal configuration policy of the module. The difference with respect to layer protocols is that these bind to components of the bottom layer that persist independently of the activities performed at the service layer, whereas business roles bind to components that are added to the service layer and have no persistency beyond the session in which they are created. Hence, in the case of the `TravelBooking` service, a new instance of `BookingAgent` is generated for each new session whereas all sessions will share the same component that binds to DB (i.e., they all share the same database of users).

The new business configuration is now easy to define:

**Definition 4.6 (The reconfigured business configuration).** In the new business configuration, the type mapping  $\mathcal{B}'$  is the same as  $\mathcal{B}$  except for the activity  $a$  for which the activity module (type)  $\mathcal{B}'(a)$  is  $\mathcal{B}(a) \oplus_{R,\rho} M$ . The homomorphism is as defined by the typing of nodes and wires discussed above.

An example of such a reconfiguration step can be given by the transition between the configurations in Figure 8 and Figure 9. As another example, the module that results from the composition indicated in Figure 18 is the one on Figure 9. This figure also illustrates the state reconfiguration step (relative to Figure 8). Notice how the component DB of the bottom layer becomes shared between the two activities ANT and LAU. The new external configuration policy is the one discussed in Section 4.1.

## 5. Related Approaches

One of the main aspects that distinguish the approach that we proposed above from other work on Web Services (e.g., [ACKM04]) and SOC in general (e.g., [OSO05]) is that we address not the middleware architectural layers (or low-level design issues in general), but what we call the ‘business level’. For instance, the main concern of the Service Component Architecture (SCA) [OSO05], from which we have borrowed concepts and notations (as discussed in more detail in Section 3), is to provide an open specification “*allowing multiple vendors to implement support for SCA in their development tools and runtimes*”. This is why SCA offers a middleware-independent layer for service composition and specific support for a variety of component implementation and interface types (e.g., BPEL processes with WSDL interfaces, or Java classes with corresponding interfaces). Our work explores a complementary direction: our research aims for a modelling framework supported by a mathematical semantics in which business activities and services

can be defined in a way that is independent of the languages and technologies used for programming and deploying the components that will execute them. The fact that the modelling framework is equipped with a formal semantics makes it possible to support the analysis of services, service compositions and activities, a direction that we are pursuing through the use of model-checking [AMFG09].

Another architectural approach to SOC has been designed [ABvH<sup>+</sup>06] that follows SCA very closely. However, its purpose is to offer a meta-model that covers service-oriented modelling aspects such as interfaces, wires, processes and data. Therefore, as in SCA, interfaces are syntactic and bindings are established at design time, whereas our interfaces are behavioural and binding occurs at run time. Other approaches to service modelling have considered richer interfaces that encompass business protocols, e.g., [BCT04, BSBM04, DD04, Rei05, Rei08], but not the dynamic aspects.

Indeed, a characteristic that distinguishes our approach from other formal models of services such as [BKM07] is the fact that we address the dynamic aspects of SOC, namely run-time discovery and binding. Formalisms for modelling (web) services tend not to address these. For example, in BPEL, service compositions are created statically and are governed by a centralised engine. This also holds for approaches that focus on choreography (e.g., [CHY07, Rei05]), where it is possible to calculate which are the partners that can properly interact with a service, but the actual discovery and binding processes are not considered. Exceptions can be found among some of the process calculi that have been developed for capturing semantic foundations of SOC (e.g., [BBC<sup>+</sup>06, BM07, LPT07]). However, such process calculi tend not to address dynamic reconfiguration separately from computation, i.e., the process of discovery and binding is handled as part of the computation performed by a service. As far as we know, SRML is the first service-modelling language to separate these two concerns.

Indeed, in our opinion, what makes SOC different from other paradigms is the fact that it concerns run-time, not design-time complexity. This is also the view exposed in [Elf07] — a very clear account of what distinguishes SOC from CBD (Component Based Development). For instance, starting from a universe of (software) components as structural entities, [BKM07] views a service as a way of orchestrating interactions among a subset of components in order to obtain some required functionality — services coordinate the interplay of components to accomplish specific tasks. Whereas in CBD component selection is either performed at design time or programmed over a fixed universe of components, SOC provides a means of obtaining functionalities by orchestrating interactions among components that are procured at run time according to given (functional) types and service level constraints.

Another area related to the work that we have presented concerns the non-functional aspects of services, namely the policies and constraints for service level agreement that have to be taken into account in the composition of services. Most of the research developed in this area has been devoted to languages for modelling specific kinds of policies (over specific non-functional features) and of selection algorithms, e.g., SCA Policy [OSO05] and several others [MDSR07, MP04, MPR<sup>+</sup>08, YL05, ea04]. These languages have been primarily designed to be part of the technology available for implementing and executing services. As such, they are tailored to the technological infrastructure that is currently enabling web services and are not appropriate for being used at high-levels of business modelling.

## 6. Concluding Remarks and Further Work

In this paper, we presented an abstract semantics of discovery and binding in service overlay computers. This is part of an effort that we started pursuing within the SENSORIA project towards a methodological and mathematical characterisation of the service-oriented computing paradigm [WHar]. The formal model that we presented is being used to provide a mathematical semantics for SRML — the SENSORIA Modelling Reference Language — on the basis of which we are defining an engineering environment that includes abstraction mappings from workflow languages (such as BPEL [BHLF07]) and policy languages (such as StPowla [BGRM08]), and model-checking techniques that support qualitative analysis [AMFG09].

Section 2 of this paper outlined methodological implications of service engineering, namely the difference between activity and service modelling, and the use of standard business protocols to facilitate service publication and delivery. A more complete account of the software engineering methodology and associated tools developed within SENSORIA for service-oriented systems can be found in [WHar]. This includes research on how elements of the UML can be specialised for supporting SOAs in general, and SRML in particular [BFL08], as well as re-engineering and model-driven development techniques.

The operational model that we proposed is based on algebraic, graph-based techniques [FLB06, FS07].

The notion of configuration and module were formalised in terms of graphs and their labelling with different kinds of components, connectors, specifications and specification morphisms. In this context, another interesting semantics of the reconfiguration process that we would like to explore is the use of graph transformations, for instance as in [BLLMT07] where the architectural style of SRML is defined.

A novel aspect of our configuration model is the way the operational semantics induces three layers that reflect different levels of change: what we called the middle or ‘service’ layer is the one that is reconfigured as a result of discovery and binding; the top layer contains components that interface with external users and launch business activities by adding new workflows to the middle layer; the bottom layer contains components that are shared by all the services executing in the middle layer and persist beyond their execution. The layers are not formally part of the architecture but help to understand the overall dynamics of the configuration of global computers and the notion of service overlay. In a sense, the bottom layer offers ‘services’ in the more standard terminology offered by component-based development (CBD) (e.g., [BKM07]). From our point of view, SOC deals with run-time, not design-time complexity, which is precisely what the middle layer intends to provide.

Discovery and selection at the service layer do not need to be programmed, but are provided by the underlying middleware, which is why they are reflected in the operational semantics defined in Section 4.1. Note that these activities operate over a set of services that is itself dynamically changing as service producers revise their portfolios. This added flexibility comes at a price — dynamic interactions have the overhead of selecting the co-party at each invocation — which means that the choice between invoking a service and calling a component needs to be carefully justified. This is why SRML makes a provision for both types of interaction (through requires and uses interfaces as discussed in Section 3), which is another feature that is unique to SRML.

A core aspect of our mathematical model is reflection as formalised in Section 2.3: business configurations superpose a typing mechanism over state configurations that identifies subsystems that execute according to given business activities and adds the corresponding types to the configuration itself. In a sense, our model provides an abstract virtual machine that captures the essence of SOC as a ‘business overlay computer’ in the sense that it abstracts away from the middleware resources e.g., the service broker and the specific platforms in which service components execute. As future work, we would like to investigate how this notion of service overlay can be implemented in reflective middleware [CBG<sup>+</sup>08].

Another interesting aspect that we propose to investigate is the ‘interleaving’ of the discovery and binding process with the execution of the activity whose workflow is being reconfigured. Having offered separate models for these two processes (the latter reported in [FLA10]), we intend to investigate how the reconfiguration process can be analysed in conjunction with the computations that are being performed by components and the coordination mechanisms on the interactions performed by the wires. Another avenue that we would like to explore in this respect is the use of graphs as a computational model, for instance as in [FHL<sup>+</sup>05].

## Acknowledgments

We would like to thank our colleagues in the SENSORIA project for many useful discussions on the topics covered in this paper (especially Emilio Tuosto for direct feedback on earlier versions), and Luís Andrade (ATX Software) in particular for his insights and suggestions on the layered approach that we proposed in the paper. We also received very useful feedback from reviewers of earlier versions of the paper. Finally, throughout the time that it took us to write and rewrite the paper, João Abreu raised many important questions on fundamental aspects of the model, which have been decisive for the version that we ended up presenting.


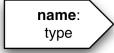


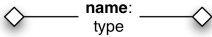


## References

- [ABFL07] João Abreu, Laura Bocchi, José Luiz Fiadeiro, and Antónia Lopes. Specifying and composing interaction protocols for service-oriented system modelling. In John Derrick and Jüri Vain, editors, *FORTE*, volume 4574 of *Lecture Notes in Computer Science*, pages 358–373. Springer, 2007.
- [ABvH<sup>+</sup>06] Wil M. P. van der Aalst, Michael Beisiegel, Kees M. van Hee, Dieter König, and Christian Stahl. A SOA-based architecture framework. In Frank Leymann, Wolfgang Reisig, Satish R. Thatte, and Wil M. P. van der Aalst, editors, *The Role of Business Processes in Service Oriented Architectures*, volume 06291 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

- [ACKM04] Gustavo Alonso, Fabio Casati, Harumi A. Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer, 2004.
- [AF08] João Abreu and José Luiz Fiadeiro. A coordination model for service-oriented interactions. In Doug Lea and Gianluigi Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.
- [AG98] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1998.
- [AMFG09] João Abreu, Franco Mazzanti, José Luiz Fiadeiro, and Stefania Gnesi. A model-checking approach for service component architectures. In Lee et al. [LLPH09], pages 219–224.
- [BBC<sup>+</sup>06] Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Thudichum Vasconcelos, and Gianluigi Zavattaro. Scc: A service centered calculus. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer, 2006.
- [BCT04] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Web service conversation modeling: A cornerstone for e-business automation. *IEEE Internet Computing*, 8(1):46–54, 2004.
- [BFGM07] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. An action/state-based model-checking approach for the analysis of communication protocols for service-oriented applications. In Stefan Leue and Pedro Merino, editors, *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2007.
- [BFL08] Laura Bocchi, José Luiz Fiadeiro, and Antónia Lopes. A use-case driven approach to formal service-oriented modelling. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 155–169. Springer, 2008.
- [BGRM08] Laura Bocchi, Stephen Gorton, and Stephan Reiff-Marganiec. Engineering service oriented applications: From StPowler processes to SRML models. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2008.
- [BHLF07] Laura Bocchi, Yi Hong, Antónia Lopes, and José Luiz Fiadeiro. From BPEL to SRML: A formal transformational approach. In Marlon Dumas and Reiko Heckel, editors, *WS-FM*, volume 4937 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2007.
- [BKM07] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1), 2007.
- [BLMT07] Roberto Bruni, Alberto Lluch-Lafuente, Ugo Montanari, and Emilio Tuosto. Service oriented architectural design. In Gilles Barthe and Cédric Fournet, editors, *TGC*, volume 4912 of *Lecture Notes in Computer Science*, pages 186–203. Springer, 2007.
- [BM07] Maria Grazia Buscemi and Ugo Montanari. Cc-pi: A constraint-based language for specifying service level agreements. In Nicola [Nic07], pages 18–32.
- [BMR97] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.
- [BS08] Simon Bliudze and Joseph Sifakis. The algebra of connectors - structuring interaction in BIP. *IEEE Trans. Computers*, 57(10):1315–1330, 2008.
- [BSBM04] Lucas Bordeaux, Gwen Salaün, Daniela Berardi, and Massimo Mecella. When are two web services compatible? In Ming-Chien Shan, Umeshwar Dayal, and Meichun Hsu, editors, *TES*, volume 3324 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 2004.
- [CBG<sup>+</sup>08] Geoff Coulson, Gordon S. Blair, Paul Grace, François Taïani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1), 2008.
- [CHY07] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Nicola [Nic07], pages 2–17.
- [DD04] Remco M. Dijkman and Marlon Dumas. Service-oriented design: A multi-viewpoint approach. *Int. J. Cooperative Inf. Syst.*, 13(4):337–368, 2004.
- [ea04] L. Zeng et al. Qos-aware middleware for web services composition. *IEEE Trans. Software Eng.*, 30(5):311–327, 2004.
- [Elf07] Ahmed Elfatatry. Dealing with change: components versus services. *Commun. ACM*, 50(8):35–39, 2007.
- [FHL<sup>+</sup>05] Gian Luigi Ferrari, Dan Hirsch, Ivan Lanese, Ugo Montanari, and Emilio Tuosto. Synchronised hyperedge replacement as a model for service oriented computing. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 22–43. Springer, 2005.
- [Fia04] José Luiz Fiadeiro. *Categories for Software Engineering*. Springer, 2004.
- [FK04] Ian T. Foster and Carl Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.
- [FLA10] José Luiz Fiadeiro, Antónia Lopes, and João Abreu. A formal model for service-oriented interactions, 2010. Available from [www.cs.le.ac.uk/srml](http://www.cs.le.ac.uk/srml).
- [FLB06] José Luiz Fiadeiro, Antónia Lopes, and Laura Bocchi. Algebraic semantics of service component modules. In José Luiz Fiadeiro and Pierre-Yves Schobbens, editors, *WADT*, volume 4409 of *Lecture Notes in Computer Science*, pages 37–55. Springer, 2006.
- [FLBAar] José Luiz Fiadeiro, Antónia Lopes, Laura Bocchi, and João Abreu. The SENSORIA reference modelling language. In Martin Wirsing and Matthias Hoelzl, editors, *Rigorous Software Engineering for Service-Oriented Systems*, LNCS. Springer, to appear.

- [FS07] José Luiz Fiadeiro and Vincent Schmitt. Structured co-spans: An algebra of interaction protocols. In Till Mossakowski, Ugo Montanari, and Magne Haveraaen, editors, *CALCO*, volume 4624 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2007.
- [GL07] Qing Gu and Patricia Lago. A stakeholder-driven service life cycle model for soa. In Elisabetta Di Nitto, Andrea Polini, and Andrea Zisman, editors, *IW-SOSWE*, pages 1–7. ACM, 2007.
- [KQCM09] David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. The Orc programming language. In Lee et al. [LLPH09], pages 1–25.
- [LLPH09] David Lee, Antónia Lopes, and Arnd Poetzsch-Heffter, editors. *Formal Techniques for Distributed Systems, Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5522 of *Lecture Notes in Computer Science*. Springer, 2009.
- [LPT07] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A calculus for orchestration of web services. In Nicola [Nic07], pages 33–47.
- [MDSR07] Arun Mukhija, Andrew Dingwall-Smith, and David S. Rosenblum. QoS-aware service composition in Dino. In *ECOWS*, pages 3–12. IEEE Computer Society, 2007.
- [MP04] Nirmal Mukhi and Pierluigi Plebani. Supporting policy-driven behaviors in web services: experiences and issues. In Marco Aiello, Mikio Aoyama, Francisco Curbera, and Mike P. Papazoglou, editors, *ICSOC*, pages 322–328. ACM, 2004.
- [MPR<sup>+</sup>08] Varvana Myllärniemi, Christian Prehofer, Mikko Raatikainen, Jilles van Gorp, and Tomi Männistö. Approach for dynamically composing decentralised service architectures with cross-cutting constraints. In Ronald Morrison, Dharini Balasubramaniam, and Katrina E. Falkner, editors, *ECSA*, volume 5292 of *Lecture Notes in Computer Science*, pages 180–195. Springer, 2008.
- [Nic07] Rocco De Nicola, editor. *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*. Springer, 2007.
- [OSO05] OSOA. Service component architecture: Building systems using a service oriented architecture, 2005. Available from [www.osoa.org](http://www.osoa.org).
- [Pah07] Claus Pahl. An ontology for software component matching. *STTT*, 9(2):169–178, 2007.
- [Pel03] Chris Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, 2003.
- [PTDL07] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40(11):38–45, 2007.
- [Rei05] Wolfgang Reisig. Modeling- and analysis techniques for web services and business processes. In Martin Steffen and Gianluigi Zavattaro, editors, *FMOODS*, volume 3535 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2005.
- [Rei08] Wolfgang Reisig. Towards a theory of services. In Roland Kaschek, Christian Kop, Claudia Steinberger, and Günther Fliedl, editors, *UNISCON*, volume 5 of *Lecture Notes in Business Information Processing*, pages 271–281. Springer, 2008.
- [RS04] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In Jorge Cardoso and Amit P. Sheth, editors, *SWSWPC*, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2004.
- [UDD04] UDDI. UDDI specification technical committee draft, 2004. Technical report, OASIS, available at [uddi.org/pubs/uddi\\_v3.htm/](http://uddi.org/pubs/uddi_v3.htm/).
- [VCS08] Hugo Torres Vieira, Luís Caires, and João Costa Seco. The conversation calculus: A model of service-oriented computation. In Sophia Drossopoulou, editor, *ESOP*, volume 4960 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2008.
- [W3C07] W3C. Simple object access protocol (SOAP), 2007. 1.2. W3C Recommendation available at [www.w3.org/TR/SOAP/](http://www.w3.org/TR/SOAP/).
- [WHar] Martin Wirsing and Matthias Hoelzl (Eds). *Rigorous Software Engineering for Service-Oriented Systems*. LNCS. Springer, to appear.
- [WLF01] Michel Wermelinger, Antónia Lopes, and José Luiz Fiadeiro. A graph based architectural (re)configuration language. In *ESEC / SIGSOFT FSE*, pages 21–32, 2001.
- [YL05] Tao Yu and Kwei-Jay Lin. A broker-based framework for qos-aware web service composition. In *EEE*, pages 22–29. IEEE Computer Society, 2005.

## Appendix A — The Iconography

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