What Do Semantics Matter When the Meat Is Overcooked?

José Luiz Fiadeiro

Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
jose@mcs.le.ac.uk

Abstract. We develop an abstract operational model for configuration management under service-oriented computing. This semantics is based on a graph-based representation of the configuration of global computers and an operational model of service-oriented dynamic reconfiguration based on a resolution-like mechanism similar to concurrent constraint programming. A resolution step involves a goal executed by a business activity and a clause that corresponds to a complex service. Unification captures service discovery, ranking and selection based on SLA-constraint optimisation and interpretations between specifications of conversations expected by the goal and provided by the discovered service. The resolvent is a reconfiguration of the original business activity that results from binding the goal with the discovered service.

1 Introduction

Given the breadth of Ugo Montanari's interests and expertise, it would not have been too difficult to contribute a paper in an area that he has touched. To match the depth of his 'touch' is, however, a much more difficult challenge. Not many people have made such profound contributions to what in computer science we usually call 'semantics', i.e. the definition of mathematical structures that explain given computational phenomena. Something that is particularly difficult in this area is to make sure that we do not obfuscate the subject of study. The title of this paper is a quote from Jan Moir, a British food critic (it is debatable whether the British know more about semantics or overcooked meat): her protest is (quite rightly) directed to sophisticated elaborations that end up destroying ingredients that would have deserved a much lighter touch. To some of us, Italian cuisine is precisely about simplicity and attention to what the products being cooked require to bring the best in them. Not surprisingly, Ugo excels in this tradition, and I am very fortunate to have been exposed to his culinary skills as a guest at a most memorable dinner that he cooked in May 2006.

In this paper, I have tried to pay tribute to Ugo by 'cooking' a 'dish' using some of the ingredients that he has cultivated (and earned fame). I should say immediately that this not a ribollita, quite the contrary: it is very much work in progress within the SENSORIA project, one of many to which Ugo has contributed during his long career. The dish is called "semantics of service-oriented configuration management".

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 that are procured through a process of discovery and negotiation that takes place at run-time, establishing a service-level agreement between the two parties. While it is recognised that specialised programming language primitives are needed that address the challenges raised by this new paradigm, we are still lacking models that are abstract enough to understand the foundations of the paradigm independently of the way services are programmed, namely from their current Web manifestation [3].

In previous papers (e.g. [2,12]), we have reported on the static aspects of SRML, a modelling language for service-oriented computing that we are defining within the SENSORIA project [20]. More precisely, we have focused on an algebraic approach for service description at the higher level of 'business modelling', and on techniques through which (simple) services can be assembled, at design-time, to create more complex services. Our contributions include language primitives for orchestrating interactions and a logic for specifying properties of conversations. Both the language and the associated logic are 'technology agnostic' in the sense that they are based on a semantic model that abstracts away from the languages in which services are programmed and the middleware that supports the coordination of interactions [1,13]. They are also expressive enough to accommodate orchestrations programmed in languages such as BPEL [7].

In this paper, we address the run-time aspects that are concerned with the way configurations of global computers change as services are discovered, selected, instantiated and bound to the applications that procured them. Once again, our aim is to provide an operational model of service-oriented configuration management that is independent of the technologies that provide the middleware infrastructure over which services can be deployed, published and discovered. For this purpose, we propose an approach inspired by (soft) concurrent constraint programming [6,19]: the process of reconfiguration is formalised in a resolution-style operational semantics that builds on the declarative algebraic semantics of SRML modelling primitives; the process of discovery, matching, ranking and selection involves unification/matching mechanisms based on c-semiring based techniques for constraint satisfaction and optimisation [5]. Familiarity with concurrent constraint programming is not strictly required as the analogy is used only for putting in context the different aspects of the operational model and its declarative semantics.

In Section 2, we lay the table by making precise what we mean by a configuration. Sections 3 and 4 address the static architectural aspects: how configurations can be structured in terms of business activities and services. Sections 5 and 6 address the dynamic aspects, i.e. how configurations change as business activities discover and bind to services. Throughout the paper, we make use of methods and techniques developed by Ugo and his colleagues. In Section 7, we say how we would like to continue doing so.

2 Configurations of Global Computers

Graphs are one of the most important commodities for any researcher working in computer science, a bit like hot (preferably boiling) water for cooks: they are not so much ingredients (i.e. they are not food as such) but enablers or domains over which one can cook a semantics. Ugo has excelled in the development and use of graph-based techniques. Our paper will also use graphs galore.

Ugo has been involved in pioneering work on the use of graphs for modelling software architectures (e.g. [15]). Because our aim is to develop a semantic domain for the way configurations of global computers are redefined as applications execute and get bound to other applications that offer required services, we choose to view configurations as graphs constituted of components (applications deployed over a given execution platform) and wires (interconnections between components over a given communication network) in a given state of execution (as in Fig. 1).

We denote by ***COMP*** and ***WIRE*** the set of all components and wires, respectively. Every component $c \in$***COMP*** and wire $w \in$***WIRE*** may be in a number of states (e.g. valuations of local state variables), the set of which is denoted by $\boldsymbol{STATE}_c$ and $\boldsymbol{STATE}_w$, respectively. We denote by ***STATE*** the corresponding indexed family of sets of states. The precise nature of these local states is of no particular importance for this paper.

A state configuration *SF* is defined to consist of:

- A simple graph G, i.e. a set *nodes(SF)* and a set *edges(SF)*; each edge *e* is associated with one and only one (unordered) pair $n \leftrightarrow m$ of nodes. We take *nodes(SF)*$\subseteq$***COMP*** (i.e. nodes are components) and *edges(SF)*$\subseteq$***WIRE*** (i.e. edges are wires).
- A (configuration) state S, i.e. an assignment of a state S$(c) \in \boldsymbol{STATE}_c$ to every $c \in$*nodes(SF)* and S$(w) \in \boldsymbol{STATE}_w$ to every $w \in$*edges(SF)*.

A state configuration <G,S> can change because either the state function S or the graph G change. We treat these two kinds of changes separately: a computation step (state change) may trigger a reconfiguration step, which needs to complete before the next computation step is performed.

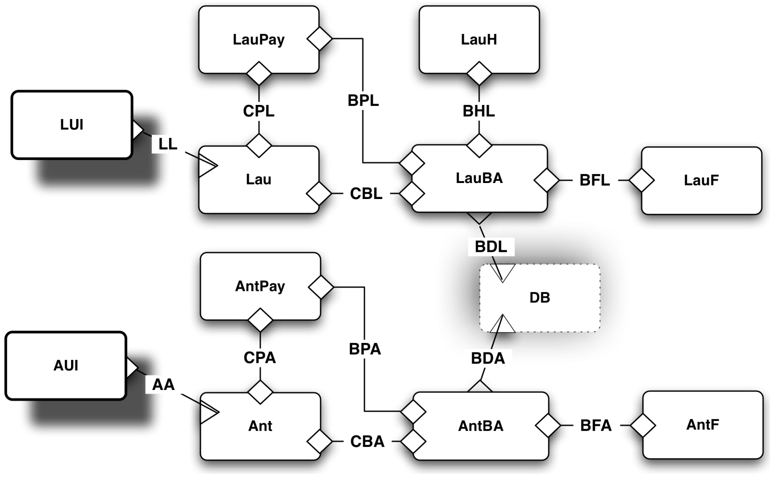

Fig. 1. The graph of a state configuration with 12 components and 13 wires

Changes to the state result from the computations executed by components and the coordination activities performed by the wires that connect them. The computational model that we are defining for SRML is explained in detail in [1]; besides providing local states for components and wires, configuration states include information on which events are pending; state transitions account for the effects of executing events on the local states and publication events. Because the configuration model that we discuss in this paper is largely independent of the computational one (except for what we call internal configuration policies below), we refrain from giving a detailed definition of that model and restrict ourselves to the aspects that are essential for understanding the way computation steps lead to reconfiguration ones. A computation step relates two state configurations $<\mathsf{G},src> \rightarrow <\mathsf{G},trg>$ by changing the state and keeping the graph invariant; reconfiguration steps, which change the graph but not the state, are considered later in the paper.

3 Services as Architectural Units

Our goal is to provide a semantics for changes in the configuration of a system, i.e. its graph, as resulting from a service-oriented architecture. More precisely, we are going to present a semantics for services as the basic units of configuration management. In this model, changes in the configuration graph – in the components that are active and the wires that connect them – result from the fact that state changes can trigger the discovery, ranking and selection of services that give rise to the addition of new components and wires that connect them to the rest of the configuration.

For this purpose, we need an architectural model of services. The basic elements of the architectural model of SRML are called modules. Together with some of his colleagues, Ugo gave a complete formalisation of the static aspects of this model in [9]. In this paper, we recall some of its essential parts and extend it to the dynamic aspects, i.e. service discovery and binding.

In Fig. 2 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).

Modules are also defined as graphs. 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, i.e. the labels of the graph are types, not instances. The types abstract from the components the business roles that they play in the activity performed by the service and, from the wires, the connectors that are responsible for coordinating the way the components interact.

Some of the nodes of a module may consist of interfaces to a pool of shared components: this is the case of ***DB*** of type *UsrDB*, i.e. a database of users. Several such "uses-interfaces" can be included in a module. Other nodes – ***PA***, ***HA***, and ***FA*** – consist of "requires-interfaces" to external services that may need to be discovered for the service to fulfil its business goal. This goal is captured in a "provides-interface" – the node ***CR*** of type *Customer*.

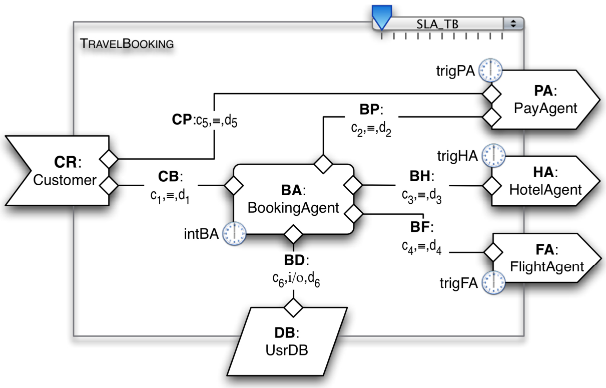

Fig. 2. The structure of a module defining the booking service of a travel agency

This notion of service module was inspired by concepts proposed in the Service Component Architecture (SCA) [21]: they are abstractions of (composite) services whose execution involves a number of interactions among coarse-grained components that perform tasks according to the underlying business logic, as well as external entities that also play a role in the business domain. These external parties are not explicitly identified in the module but only implicitly through what we have called external-interfaces. External 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 remote 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.

The operational model that we wish to present for configuration management is independent of the formalisms used for defining business roles, business protocols, layer protocols and connectors. Therefore, we will not discuss these formalisms in the paper; we assume instead that we have available sets ***BROL***, ***BUSP*** and ***LAYP*** of specifications of business roles, business protocols and layer protocols, respectively (see [12] for an overview of the formalisms used in SRML).

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 shared components that persist independently of the activities performed by the services, whereas business roles bind to components that are created when the session starts and have no persistency beyond that session. 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 *UsrDB* (i.e. they all share the same database of users).

Connectors, which label wires, are triples <μ_A,*P*,μ_B> where:

- *P* is an "interaction protocol." Every interaction protocol has two roles $roleA_P$ and $roleB_P$, and a glue $glue_P$. The glue is a description of the coordination mechanisms enforced by the protocol, which we assume to be given in a formalism ***IGLU***.
- μ_A and μ_B are 'attachments' that connect the roles of the protocol to the entities (business roles or protocols) being interconnected.

Interaction protocols are often just straight connections between ports identifying which interactions in *roleA* correspond to which interactions in *roleB*. In many cases, the interaction glue may include the routing of events, encryption/decryption of messages, or transforming sent data to the format expected by the receiver. In a connector, the interaction protocol is bound to the parties via attachments: these are mappings from the roles to the signatures of the parties identifying which interactions of the parties perform which roles in the protocol. We use ***CNCT*** to designate the set of connectors. See [2] for a more detailed account of how connectors are formalised in SRML. In software architecture, one can define connectors that involve an arbitrary number of roles, but service-oriented architectures involve only interactions between two partners.

In addition to a graph, a module identifies two important aspects related to the way a service can change a configuration:

- An internal configuration policy (indicated by the symbol 🕓) that identifies the triggers of the external service discovery process, and the initialisation and termination conditions of the components.
- An external configuration policy (indicated by the symbol SLA) that consists of the variables and constraints that determine the quality profile to which the discovered services need to adhere.

The configuration policies (both internal and external) are discussed below together with a formal definition of the notion of module.

4 Business Configurations

As already explained, we approach the operational aspects of SOC from the point of view of the execution of business processes: our aim is to see state configurations as a result of the joint execution of a number of activities that can trigger the discovery and binding of external services. In the previous section, we discussed how services define architectural units. In this section, we discuss how the configuration itself is structured so that these units can be plugged together. This configuration structure is given by what we call business activities.

We take business activities to be characterised, in every configuration, by

- A sub-configuration, i.e. a subset of the components, and the wires between them, that execute as part of the activity.
- A workflow that implements the "business logic" of the activity.

For instance, we would like to recognise two activities in Fig. 1 whose sub-configurations are as depicted in Fig. 3. Intuitively, both correspond to two instances of the same business logic (two costumers booking their travel) but at different stages of their workflow: one (launched by *LUI*) is already connected to a flight and a hotel agent (*LauF* and *LauH*, respectively) but the other (launched by *AUI*) is also connected to a (different) flight agent (*AntF*) still has to find a hotel agent. Both share a database *DB* (of users), which is a persistent component.

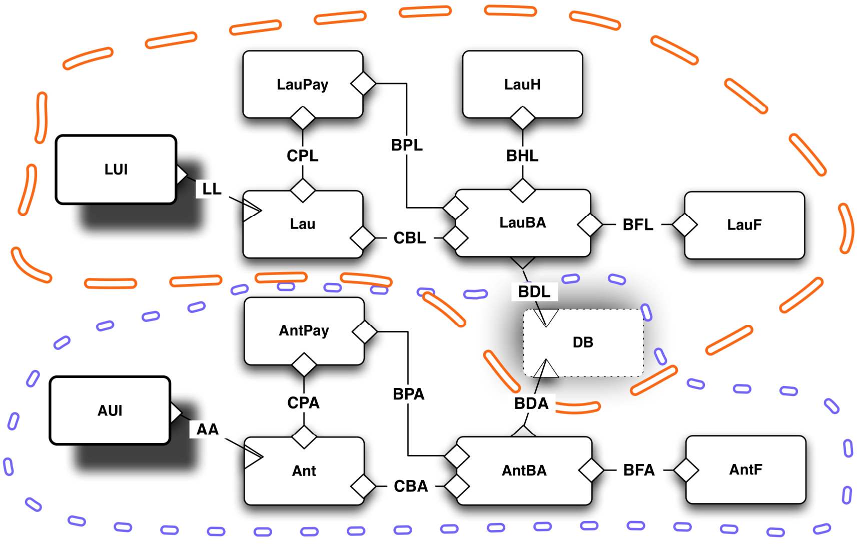

Fig. 3. The sub-configurations corresponding to two business activities

What we are calling ‘business workflow’ is formally captured by typing the sub-configuration of the activity by what we call an ‘activity module’. For instance, the activity module depicted in Fig. 4 types some of the components of the configuration depicted in Fig. 1 with business roles and some of its wires with connectors.

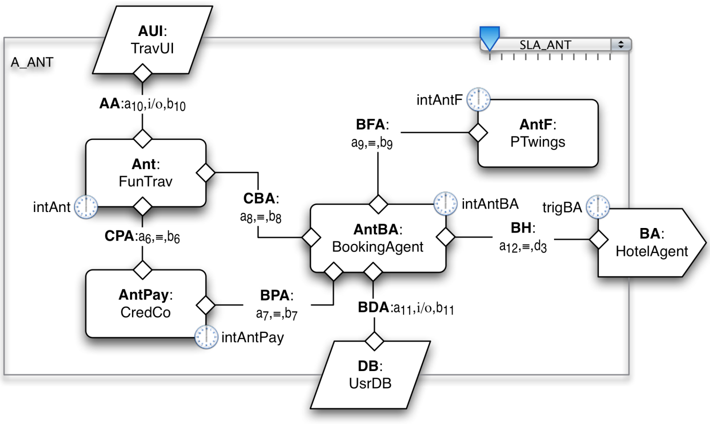

Fig. 4. An activity module

Activity modules are like service modules except that, instead of a provides-interface, they include a 'serves-interface' through which users can interact with the activity. This is the case of ***AUI*** of type *TravUI* – a user interface for travel booking. Like uses-interfaces, serves-interfaces are labelled by layer protocols. It is important to understand the difference between serves- and provides-interfaces. In the case of the provides-interface, the corresponding party is the customer to which the module will be bound to provide a service. This customer is the business activity that triggered the discovery of the service, not the top layer user. The latter binds to the serves-interface of the activity. Hence, in the case of the business configuration depicted in Fig. 3, we can see components – ***Lau*** and ***Ant*** – that interact with the users of the activities through two interfaces (***LUI*** and ***AUI***, respectively).

In summary, a module *M* is defined to consist of:

- A graph *graph(M)*.
- A distinguished subset of nodes $requires(M) \subseteq nodes(M)$.
- A distinguished subset of nodes $uses(M) \subseteq nodes(M)$.
- In the case of service modules, a node $provides(M) \in nodes(M)$ distinct from *requires(M)* and *uses(M)*.
- In the case of activity modules, a node $serves(M) \in nodes(M)$ distinct from *requires(M)* and *uses(M)*.
- We denote by components(M) the set of nodes(M) that are not provides(M) or serves(M), nor in requires(M) or uses(M).
- We denote by *body(M)* the (full) sub-graph of *graph(M)* that consists of *components(M)* and all the edges between them.
- A labelling function $label_M$ such that
  - o $label_M(n) \in \boldsymbol{BROL}$ if $n \in components(M)$
  - o $label_M(n) \in \boldsymbol{BUSP}$ if $n \in provides(M) \cup requires(M)$
  - o $label_M(n) \in \boldsymbol{LAYP}$ if $n \in serves(M) \cup uses(M)$
  - o $label_M(e{:}n \leftrightarrow m)$ is a connector $<\mu_A, P, \mu_B>$ such that μ_A (resp. μ_B) is an attachment between $roleA_P$ and $label_M(n)$ (resp. $roleB_P$ and $label_M(m)$).
- An internal configuration policy (see below)
- An external configuration policy (see below)

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:

- Each requires-node $n \in requires(M)$ has an associated trigger condition *trigger(n)*: this is a condition that is evaluated over the state of the configuration. 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 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.

- Each component-node $n \in components(M)$ has an initialisation condition *init(n)* that is ensured when the component is instantiated. That is, if we have a computation step $<\mathsf{G},src>\rightarrow<\mathsf{G},trg>$ followed by a reconfiguration step $<\mathsf{G},trg>\rightarrow<\mathsf{H},trg'>$, the state *trg'* must coincide with *trg* on the components and wires that are carried over to H as well as ensure that *init(n)* holds for any new node *n* in H.
- Each component-node has a second state condition *term(n)* that determines when the component stops executing and interacting with the rest of the components of the activity. That is, if a computation step $<\mathsf{G},src>\rightarrow<\mathsf{G},trg>$ is such that *term(n)* holds of a component *n* in state *trg* then *n* can be removed from G in the subsequent reconfiguration step.

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 and standard configuration parameters selected from a fixed set – *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 SRML, we adopt the framework for constraint satisfaction and optimization defined by Ugo and colleagues in [5], in which constraint systems are defined in terms of c-semirings. As explained therein, this framework is quite general and allows us to work with constraints of different kinds – both hard and 'soft', the latter in many grades (fuzzy, weighted, and so on). The c-semiring approach also supports selection based on a characterisation of 'best solution' supported by multi-dimensional criteria, e.g. minimizing the cost of a resource while maximizing the work it supports. See [10] for other usages of this approach for service ranking and selection.

In summary, an external configuration policy consists of:

- A constraint system $cs(M)=\langle S,D,V\rangle$ where S is a c-semiring, V is a totally ordered set (of configuration variables), and D is a finite set (domain of possible elements taken by the variables).
- A set *sla(M)* of constraints over *cs(M)*; every constraint consists of a selected subset *con* of variables and a mapping $def{:}D^{|con|}\rightarrow S$ that assigns a degree of satisfaction to each tuple of values.
- For every variable in *cs(M)*, an *owner* – either a node or an edge of *M*.

We can now define the notion of business configuration that accounts for the coarser business dimension that is overlaid by services on state configurations. We presuppose a space A of business activities. We also assume that we have typing relations ***COMP×BROL***, ***COMP×LAYP*** and ***WIRE×CNCT*** through which we can tell whether a given component (resp. wire) complies with a given business role or layer protocol (resp. connector).

A business configuration consists of:

- A state configuration *SF*.
- A partial mapping B that assigns a module $\mathsf{B}(a)$ to the activities $a \in \mathsf{A}$ that are active in *SF* – the workflow being executed by *a* in the configuration *SF*.
- A mapping C that assigns an homomorphism $\mathsf{C}(a)$ of graphs $body(\mathsf{B}(a)) \rightarrow SF$ to every activity $a \in \mathsf{A}$ that is active in *SF*. This homomorphism types the nodes of the activity with business roles or layer protocols – i.e. $\mathsf{C}(a)(n){:}label_{\mathsf{B}(a)}(n)$ for every node *n* – and the edges with connectors – i.e. $\mathsf{C}(a)(e){:}\ label_{\mathsf{B}(a)}(e)$ for every edge *e* of the body of the activity.

The homomorphism labels the components and wires of the state configuration with the business elements (business roles and the connectors) that they fulfil in the activity.

5 Services as Clauses

The operational semantics that we wish to put forward for service-oriented reconfiguration is inspired by another of Ugo's contributions to computer science, this time to concurrent constraint programming (CCP) in its 'soft' version [6]. More precisely, our approach is not CCP *sensu stricto*: it borrows aspects and techniques from CCP but it also adds a few (interesting) new ingredients.

The analogy starts with the identification of every service module with a 'clause':

$$P \xleftarrow[body(M)]{} R_1, \ldots, R_n$$

where $label_M(provides(M))=P$ and $label_M(requires(M))=\{R_1,\ldots,R_n\}$. In logic programming "speak", the clause states that, to obtain *P*, one has to find $R_1,\ldots,R_n$ and execute *body(M)*. The execution of the body corresponds, in a sense, to the computation that, in the execution of a Horn clause, is performed to provide an 'answer' – what in logic programming corresponds to a substitution.

Using this representation, a business activity *a* is of the form:

$$\xleftarrow[B(a)]{} R_1, \ldots, R_n$$

where *B(a)* is $body(\mathsf{B}(a))$. That is, a business activity corresponds to a goal clause: finding $R_1,\ldots,R_n$ and providing an answer through the execution of *B(a)*.

Like in concurrent logic programming (CLP), we are not interested in the "don't know" ("angelic") non-determinism that results from exploring, through backtracking, all possible alternative matches to the R_i: if we are not happy with the chosen service provider, we cannot go back in time and restart with another provider! That is, we 'commit' to the choice of service provider. We deal instead with what is sometimes called "indeterminism" (or "don't care" non-determinism), which results from the existence of a choice of service provider. In CLP, this choice is controlled by the 'guard' assigned to each clause – a sequence of goals that appears before the body of the clause, which need to be executed successfully for the clause to be chosen and the body to be executed. Among all clauses that have satisfiable guards, one of them is chosen and the execution 'commits' to it.

In CCP [19], one works in a more general setting in which all processes executing can interact by means of a shared set of constraints to which they can add new constraints ('tell') or check if they entail a given constraint ('ask'). Soft CCP [6] generalises these mechanisms even further by working over a c-semiring as used in Section 0 for external configuration policies: one can then choose among all satisfiable clauses that maximise the degree of satisfaction relative to the set of constraints. In our setting, this means that we work instead with clauses of the form:

$$P \xleftarrow[body(M)]{} sla(M) \mid R_1, \ldots, R_n$$

where the set *sla(M)* acts as a 'soft-guard'.

As explained later in the paper, when the discovery of a requires-interface R_i of an activity

$$\xleftarrow[B(a)]{} sla(a) \mid R_1, \ldots, R_n$$

is triggered, the matching process identifies service modules (clauses) *M*

$$P \xleftarrow[body(M)]{} sla(M) \mid T_1, \ldots, T_m$$

and a 'unifier'-morphism ρ that is an interpretation between R_i and *P*, and makes the combination $sla(B(a)) \oplus_{R,\rho} sla(M)$ of the sets of constraints of *B(a)* and *M* consistent.

The selection of the clause (service provider) is made among those that maximise the degree of satisfaction of the combined set of constraints. The resolvent is another goal clause corresponding to the reconfiguration of the business activity *a*:

$$\xleftarrow[B'(a)]{} sla'(a) \mid R_1, \ldots, R_{i-1}, T_1, \ldots, T_m, R_{i+1}, \ldots, R_n$$

where $B'(a)=body(B(a) \oplus_{R,\rho} M)$ and $sla'(a)=contract(sla(B(a)) \oplus_{R,\rho} sla(M))$ as defined below. That is, *B'(a)* is the body of the new workflow of the activity *a* that results from the binding with the discovered service and *sla'(a)* is the contract negotiated between *a* and *M*, which extends the amalgamated set of constraints of both *a* and *M*. That is, from the point of view of CCP, new constraints are added to the current set of the activity (each activity has its own set of constraints and its execution interferes with other activities only through the shared persistent components).

When a state is reached in which the activity *a* is an empty clause of the form

$$\xleftarrow[B(a)]{} sla(a)$$

the resolution process for that activity will have ended, meaning that the activity does not need any external services and will continue executing according to the same workflow until completion (though one may simplify the configuration by removing components as they finish executing). By then, all relevant quality-of-service variables will have been instantiated according to *sla(a).*

However, one may not need to discharge all the requires-interfaces (i.e. bind them to service providers). The resolution step does not spawn immediately all the body goals in parallel; instead, we wait for the triggers declared for each goal T_i (in the example above) to become true in order to launch the corresponding discovery, ranking and selection process. Notice that the evaluation of the triggers will change as execution proceeds. This is because the body *B(a)* of the activity will be executing and changing the state over which the triggers are evaluated, and *sla(a)* will itself change as new constraints are added. From the point of view of concurrent programming, this means that the goals are guarded by their triggers, i.e. they are of the form *(**ask**(trigger(n_i))*$\rightarrow$*R_i)* where n_i is the node labelled by R_i and ***ask**(c)* checks if condition *c* is entailed by the current state and set of constraints.

Because the state may change, what matters is not so much the consistency of the trigger with the current state and sla, but the fact that it may become true in a future state. However, contrarily to CCP, we do not take non-satisfiability as failure. As seen in Section 0, the internal configuration policy contains a termination condition for each component interface that determines when the execution of the instances should stop. For any *(**ask**(trigger(n_i))*$\rightarrow$*R_i)* still outstanding when all components have terminated, the condition *trigger(n_i)* will not be satisfiable in time, which we do not consider to be a failure. Therefore, the components that are delivering the service may finish executing their (distributed) workflow without having triggered all the conditions in the service internal configuration policy.

6 Reconfiguration as Resolution

In logic programming, different strategies may be adopted for choosing the next query to be processed, which in our case means the next service to be discovered. In our model, this choice is given by the occurrence of triggers. As mentioned in Section 0, every module declares, as part of its internal configuration policy, the triggering conditions that apply to their requires-interfaces. Given a business configuration $BC=\langle<\mathcal{G},\mathcal{S}>,\mathcal{B},\mathcal{C}\rangle$ and an activity *a*, each condition *trigger(r)* is evaluated over the state $\mathcal{S}$. If the condition *trigger(R)* for a given requires-interface *R* holds in *BC*, the “unification” process is launched, which should return a service that “best” fits the business protocol $label_{\mathcal{B}(a)}(R)$ and the external configuration policy of $\mathcal{B}(a)$.

In our setting, this unification process involves three steps, which we can outline 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* – 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. Select one of the services that maximises the level of satisfaction offered by the corresponding contract.

We are now going to define each of these steps in more detail, though most of the technical aspects need to be consulted in [5] and [13,14]. Consider a business configuration $BC=\langle SF,\mathcal{B},\mathcal{C}\rangle$ and let *R* be a requires-interface of a business activity *a* such that *trigger(R)* holds in *SF*. The discovery phase returns all the service modules *M* that satisfy the following properties:

- There is a specification morphism ρ: $label_{\mathcal{B}(a)}(R)\rightarrow label_M(provides(M))$, i.e. the behavioural properties offered by the provides interface of the candidate service module 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(B(a))*. This means that we can extend the mapping ρ in such a way that, for every variable *v* in *cs(B(a))*:
- if owner(v)=R, there exists ρ(v) in cs(M) such that type(v)=type'(ρ(v)) and owner'(ρ(v))=provides(M);
- if *owner(v)* is a wire $i \leftrightarrow R$ then, for every wire *w'* in *M* of the form *provides(M)*$\leftrightarrow$*j*, there is a variable *ρ(v,w')* in *cs(M)* s.t. *owner'(ρ(v,w'))=w'* and *type(v)=type'(ρ(v,w'))*.
- The combination $sla(B(a)) \oplus_{R,\rho} sla(M)$ of the sets of constraints of *B(a)* and *M* is consistent (as defined below).

Intuitively, compatibility means that each discovered service needs to support the properties required by the activity through the business protocol associated with *R* and 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 first condition (entailment of properties) is handled through the logic that is adopted for specifying business protocols (see [12] for a flavour of the logic used in SRML). The second condition ensures that is indeed possible to achieve a service-level agreement between the activity and the service module. Compatibility of the constraint systems of *B(a)* and *M* relative to *R* ensures that they can be combined, which gives rise to another constraint system.

The combined constraint system $cs(B(a)) \oplus_{R,\rho} cs(M)$ is defined as follows:

- Its domain *D"* is the union $D \cup D'$ of the domains of *cs(B(a))* and *cs(M)*.
- Its set of variables *V* is the disjoint union of *cs(B(a))* and *cs(M)* except for all pairs *v*|*ρ(v)* and *v*|*ρ(v,w')*, which give rise to variables (those involved in the negotiation). Notice that, if *owner(v)* is a wire $i \leftrightarrow R$, then we may end up with several "aliases" *v*|*ρ(v,w')*, one for each wire *w'* in *M* of the form *provides(M)*$\leftrightarrow$*j*. We denote by *neg(R,ρ)* the set of such variables.

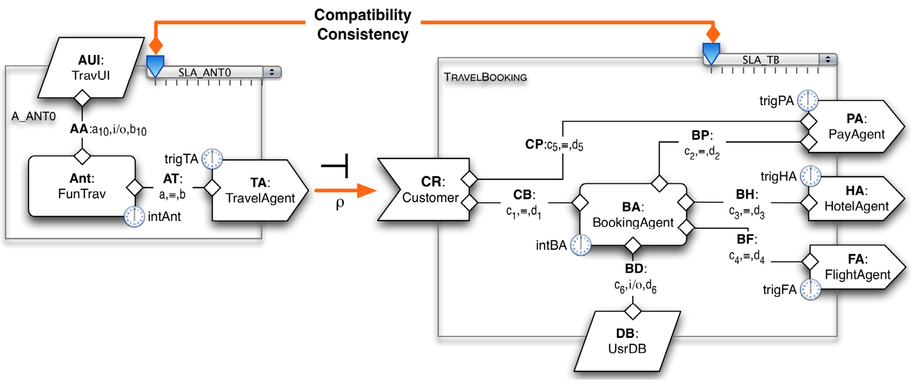

Fig. 5. The elements involved in unification

The combined set of constraints $sla(\mathcal{B}(a))\oplus_{R,\rho}sla(M)$ is defined by 'lifting' the constraints of $sla(\mathcal{B}(a))$ and $sla(M)$ to the new constraint system.

In order to illustrate these constructions, consider that, at a certain point of the execution of the workflow of the business activity *A_ANT0* in **Fig. 5**, the condition *trigTA* becomes true and triggers the unification process for *TA*. For the service module *TRAVELBOOKING* 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.

Finally, we can discuss how contracts are established. Together with the set $neg(R,\rho)$ of the variables being negotiated (those in the domain of ρ), the set of constraints $sla(\mathcal{B}(a))\oplus_{R,\rho}sla(M)$ defines a constraint problem. In the c-semiring framework, 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 $neg(R,\rho)$. Ranking a discovered service M in our framework consists in finding an assignment that maximizes the degree of satisfaction. The constraint that results from the negotiation is denoted by $contract(\mathcal{B}(a)\oplus_{R,\rho}M)$. The selected service is one with maximal rank.

It remains to define the new business configuration that results from the process of discovery, ranking and selection – what we could call the 'resolution step' using the analogy with logic programming. 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. Consider that a service module M is 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 $\mathcal{B}(a)\oplus_{R,\rho}M$ (depicted in Fig. 6 for *A_ANT0* and *TRAVELBOOKING*) defined as follows:

- 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 adding an edge $i\leftrightarrow j$ between any two nodes i and j such that $i\leftrightarrow R$ is an edge of $\mathcal{B}(a)$ and $provides(M)\leftrightarrow j$ is an edge of M. 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 execution plan 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 morphism ρ. These are calculated by composing the connectors that label $i \leftrightarrow R$ and $provides(M) \leftrightarrow j$. This process of composition is detailed in [13]: basically, we need to compose the glues of the connectors through the roles that have been bound through the signature morphism.

- The external configuration policy is *contract*($\mathcal{B}(a) \oplus_{R,\rho} M$) and the triggers, initialisation and termination conditions of the internal configuration policy are all inherited from $\mathcal{B}(a)$ and M.

We take this module to provide the reconfigured execution plan of the business activity *a*. We can now define the new state and business configurations that result from the discovery and binding processes. 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 the shared persistent components, which are typed by the layer protocols of *uses(M)*. Notice that we do not create new shared persistent components (instances) in this process: such components are used, not created by services.
- The new components and wires are initialised so as to satisfy the internal configuration policy of *M*.

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

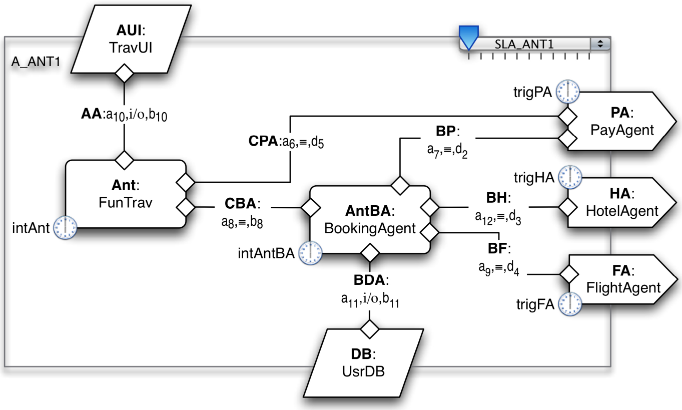

Fig. 6. A new session of *TravelBooking* starts and reconfigures the workflow of *ANT*

7 Final Tasting

We cooked a dish using some of the ingredients that Ugo has given us during his career. As any good amateur of Italian cuisine would have done, the chosen ingredients are of top quality. As for the dish, "provate per credere"…

We certainly hope not to have 'overcooked the meat'. In our opinion, widely shared within the SENSORIA project, there is a lot of research that needs to be done towards a methodological and mathematical characterisation of the service-oriented computing paradigm [20]. Our approach differs from other work on Web Services (e.g. [3]) and SOC in general (e.g. [21]) in that we address not the middleware architectural layers or low-level design issues, but what we call the 'business level'. That is, we view SOC as operating over configurations of global computers that are typed by business activities, which may need to discover and bind to external services as they execute and, therefore, reconfigure their activity.

This emphasis on the business dimension is well apparent in the semantic model that we proposed in the sense that it separates the reconfiguration (business level) from the computation dimension (state level). More specifically, our model makes only minimal assumptions about the computational aspects that account for state changes and interactions, as well as the languages and formalisms that are used for specifying the workflows executed by components, the interaction protocols established through the wires, and the properties that describe the properties of services. The specific formalisms used in the SENSORIA Reference Modelling Language (SRML) are presented in [2,12]. Other popular formalisms for modelling (web) services are those also adopted for business workflows [17,18], as well different kinds of process calculi (e.g. [8,10,16]). However, the workflow-oriented formalisms tend not to address dynamic reconfiguration and the process calculi tend not to address it separately from computation. As far as we know, SRML is the first service-modelling language to separate these two concerns.

The semantics of the actual reconfiguration operated during a resolution step was given based on algebraic, graph-based techniques [13,14]. 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 [9] where the architectural style of SRML has been defined by Ugo and some of his colleagues.

Another aspect worth investigating is the "interleaving" of the synthesis/resolution process with the execution of the activity whose workflow is being synthesised. Having offered separate models for these two processes, 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. For this purpose, we will rely on calculi (e.g. [16]) and logics (e.g. [4]) that are being developed within SENSORIA. Another avenue that we would like to explore in this respect is the use of graphs as a computational model, for instance as developed in [11], once more with Ugo's contribution.

Acknowledgments

The reviewers kindly suggested many improvements and extensions, which unfortunately would not fit in the space available for the paper. Most of the work reported in this paper was developed in collaboration with Antónia Lopes, Laura Bocchi, and João Abreu. I would like to thank our colleagues in the SENSORIA project for many useful discussions on the topics covered in this paper, which includes not only Ugo but also many of Ugo's "children" (and "grandchildren"). This is the last contribution to computer science that I would like to thank Ugo for: to have populated the area with so many brilliant scientists (though their performance as cooks still needs to be properly tested...).

References

1. Abreu, J., Fiadeiro, J.: A coordination model for service-oriented interactions. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 1–16. Springer, Heidelberg (2008)
2. Abreu, J., Bocchi, L., Fiadeiro, J.L., Lopes, A.: Specifying and composing interaction protocols for service-oriented system modelling. In: Derrick, J., Vain, J. (eds.) FORTE 2007. LNCS, vol. 4574, pp. 358–373. Springer, Heidelberg (2007)
3. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services. Springer, Heidelberg (2004)
4. ter Beek, M., Fantechi, A., Gnesi, S., Mazzanti, F.: An action/state-based model checking approach for the analysis of communication protocols for Service-Oriented Applications. In: Formal Methods for Industrial Critical Systems. LNCS, Springer, Heidelberg (to appear)
5. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. Journal of the ACM 44(2), 201–236 (1997)
6. Bistarelli, S., Montanari, U., Rossi, F.: Soft concurrent constraint programming. ACM Transactions on Computational Logic 7(3), 563–589 (2006)
7. Bocchi, L., Hong, Y., Lopes, A., Fiadeiro, J.: From BPEL to SRML: a formal transformational approach. In: Dumas, M., Heckel, R. (eds.) Web Services and Formal Methods. LNCS, vol. 4937, pp. 92–107. Springer, Berlin, Heidelberg, New York (2008)
8. Boreale, M., et al.: SCC: a service centered calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Berlin, Heidelberg, New York (2006)
9. Bruni, R., Lluch Lafuente, A., Montanari, U., Tuosto, E.: Service oriented architectural design. In: Trustworthy Global Computing, Springer, Berlin, Heidelberg, New York (to appear, 2007)
10. Buscemi, M., Montanari, U.: CC-Pi: A constraint-based language for specifying service level agreements. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 18–32. Springer, Berlin, Heidelberg, New York (2007)
11. Ferrari, G.F., Hirsch, D., Lanese, I., Montanari, U., Tuosto, E.: Synchronised hyperedge replacement as a model for service oriented computing. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 22–43. Springer, Berlin, Heidelberg, New York (2006)

12. Fiadeiro, J.L., Lopes, A., Bocchi, L.: A formal approach to service-oriented architecture. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 193–213. Springer, Berlin, Heidelberg, New York (2006)
13. Fiadeiro, J.L., Lopes, A., Bocchi, L.: Algebraic semantics of service component modules. In: Fiadeiro, J.L., Schobbens, P.-Y. (eds.) WADT 2006. LNCS, vol. 4409, pp. 37–55. Springer, Berlin, Heidelberg, New York (2007)
14. Fiadeiro, J.L., Schmitt, V.: Structured co-spans: an algebra of interaction protocols. In: Mossakowski, T., Montanari, U., Haveraaen, M. (eds.) CALCO 2007. LNCS, vol. 4624, pp. 194–200. Springer, Berlin, Heidelberg, New York (2007)
15. Hirsch, D., Montanari, U.: Two graph-based techniques for software architecture reconfiguration. Electronic Notes in Theoretical Computer Science 51, 177–190 (2001)
16. Lapadula, A., Pugliese, R., Tiezzi, F.: Calculus for orchestration of web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Berlin, Heidelberg, New York (2007)
17. Ouyang, C., Verbeek, E., van del Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Science of Computer Programming 67(2-3), 162–198 (2007)
18. Reisig, W.: Modeling and analysis techniques for web services and business processes. In: Steffen, M., Zavattaro, G. (eds.) FMOODS 2005. LNCS, vol. 3535, pp. 243–258. Springer, Berlin, Heidelberg, New York (2005)
19. Saraswat, V.A.: Concurrent Constraint Programming. MIT Press, Cambridge, Massachusetts (1993)
20. SENSORIA consortium (2007), `http://www.sensoria-ist.eu/files/whitePaper.pdf`
21. The Open Service Oriented Architecture collaboration, `http://www.osoa.org`