Algebraic Semantics of Service Component Modules[†]

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

[1] Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
{bocchi,jose}@mcs.le.ac.uk

[2] Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa, PORTUGAL
mal@di.fc.ul.pt

Abstract. We present a notion of module acquired from developing an algebraic framework for service-oriented modelling. More specifically, we give an account of the notion of module that supports the composition model of the SENSORIA Reference Modelling Language (SRML). The proposed notion is independent of the logic in which properties are expressed and components are programmed. Modules in SRML are inspired in concepts proposed for Service Component Architecture (SCA) and Web Services, as well the models that have been proposed for Algebraic Specifications, namely by H. Ehrig and F. Orejas, among others; they include interfaces for required (imported) and provided (exported) services, as well as a number of components (body) whose orchestrations ensure how given behavioural properties of the provided services are guaranteed assuming that the requested services satisfy required properties.

1 Introduction

In the emerging service-oriented computing paradigm, services are understood as autonomous, platform-independent computational entities that can be described, published, discovered, and dynamically assembled for developing massively distributed, interoperable, evolvable systems. In order to cope with the levels of complexity entailed by this paradigm, one needs abstractions through which complex systems can be understood in terms of compositions of simpler units that capture structures of the application domain. This is why, within the IST-FET Integrated Project SENSORIA – *Software Engineering for Service-Oriented Overlay Computers* – we are developing an algebraic framework for supporting service-oriented modelling at levels of abstraction that are closer to the "business domain".

More precisely, we are defining a suite of languages that support different activities in service-oriented modelling to be adopted as a reference modelling "language"

[†] This work was partially supported through the IST-2005-16004 Integrated Project *SENSORIA: Software Engineering for Service-Oriented Overlay Computers*, and the Marie-Curie TOK-IAP MTK1-CT-2004-003169 *Leg2Net: From Legacy Systems to Services in the Net*.

– SRML – within the SENSORIA project. In this paper, we are concerned with the "composition language" SRML-P through which service compositions can be modelled in the form of business processes, independently of the hosting middleware and hardware platforms, and the languages in which services are programmed. The cornerstone of this language is the notion of *module* through which one can model composite services understood as services whose business logic involves the invocation of other services.

In our approach, a module captures a business process that interacts with a set of external services to achieve a certain "goal". This goal should not be understood as a "return value" to be achieved by a computation in the traditional sense, but as a "business interaction" that is offered for other modules to discover and engage with. Global business goals emerge not from prescribed computations but from the peer-to-peer, conversational interactions that are established, at run-time, between business partners. This is why software development in the service-oriented paradigm requires new abstractions, methods and techniques.

The challenge that we face, and on which we wish to report, is to support this paradigm with mathematical foundations that allow us to define, in a rigorous and verifiable way, (1) the mechanisms through which modules can use externally procured services to offer services of their own, and (2) the way modules can be assembled into (sub-)systems that may, if desired, be offered themselves as (composite) modules. Having this goal in mind, we present in Section 2 a brief overview of the supported composition model and a summary of the different formal domains involved in it. Then, in Section 3, we formalise the notion of module as a graph labelled over the identified formal domains. Section 4 discusses the correctness property of modules and the notion of system as an assembly of modules. Finally, Section 5 develops the notion of composition through which composite modules are defined from systems.

2 The Composition Model

Modules in SRML-P are inspired by concepts proposed in Service Component Architectures (SCA) [7]. The main concern of SCA is to develop a middleware-independent architectural layer that can provide an open specification "allowing multiple vendors to implement support for SCA in their development tools and runtimes". That is, SCA shares with us the goal of providing a uniform model of service behaviour that is independent of the languages and technologies used for programming and deploying services. However, whereas we focus on the mathematical structures that support this new architectural model, SCA looks "downstream" in the abstraction hierarchy and offers specific support for a variety of component implementation and interface types such as BPEL processes with WSDL interfaces, and Java classes with corresponding interfaces.

Given the complementarities of both approaches, we decided to stay as close as possible to the terminology and methodology of SCA. This is why in SRML-P we adopt the following formal domains when characterising the new architectural ele-

ments: business roles that type SCA *components*, business protocols that type SCA *external interfaces* (both entry points and required services), and interaction protocols that type SCA *internal wires*.

Service components do not provide any business logic: the units of business logic are *modules* that use such components to provide services when they are interconnected with a number of other parties offering a number of required services. In a SRML-P module, both the provided services and those required from other parties are modelled as *external interfaces*, or interfaces for short. Each such interface specifies a stateful interaction between a service component and the corresponding party, i.e. SRML-P supports both "syntactic" and "behavioural" interfaces.

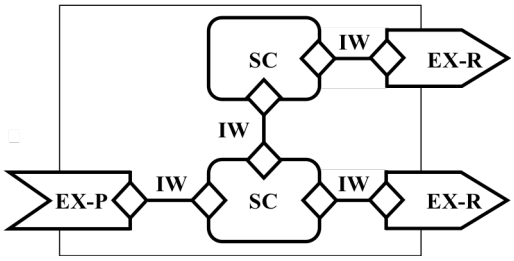


Figure 1: A SRML-P module; SC–service component; EX-P–(provides) external interface; EX-R–(requires) external interface; IW–internal wire

The service components within a module orchestrate the interactions with the external parties that, in any given configuration, are linked to these interfaces. Like in SCA, modules are interconnected within systems by linking required external services of given modules with provided services offered by other modules. Such interconnections can be performed "just-in-time" once a mechanism is provided through which modules can be "discovered" and the binding of required with provided external interfaces can be effectively supported.

2.1 Business roles

Central to SCA is the notion of *component*. In SRML-P, a component is a computational unit that fulfils a given *business role*, which is modelled in terms of an execution pattern involving a number of interactions that the component can maintain with other parties. We refer to the execution pattern as an *orchestration element*, or *orchestration* for short.

The model provided through the business role is independent of the language in which the component is programmed and the platform in which it is deployed; it may be a BPEL process, a Java program, a wrapped-up legacy system, *inter alia*. The orchestration is independent of the specific parties that are actually interconnected with the component in any given run-time configuration; a component is totally independent in the sense that it does not invoke services of any specific co-party – it just offers an interface of two-way interactions in which it can participate.

The primitives that we are adopting in SRML-P for describing business roles have been presented in [4] and, in more detail, also in [3]; they are defined in terms of typical event-condition-action rules in which the actions may involve interactions with other parties. An example is given in the appendix in terms of a BookingAgent of a typical TravelBooking composite service. However, given that our focus in this paper is the notion of module, we do not need to commit to any specific orchestration language and, therefore, will not discuss the language used in SRML-P any further. All we need is to assume a set ***BROL*** of business roles to be given together with a number of mappings to other formal domains as detailed further on.

2.2 Signatures

One of the additional formal domains that we need to consider consists of the structures of interactions through which components can be connected to other architectural elements. These structures capture both classical notions of "syntactic" interface – i.e. declarations of types of interactions – and the ports through which interconnections are established. In SRML-P, interactions can be typed according to the fact that they are synchronous or asynchronous, and one or two-way; parameters can also be defined for the exchange of data during interactions.

We assume that such structures are organised in a category ***SIGN***, the objects of which are called *signatures*. Morphisms of signatures define directional "part-of" relationships, i.e. a morphism $\sigma: S_1 \rightarrow S_2$ formalises the way a signature (structure of interactions) S_1 is part of S_2 up to a possible renaming of the interactions and corresponding parameters. In other words, a morphism captures the way the source is connected to the target, for instance how a port of a wire is connected to a component.

We assume that every business role defines a signature consisting of the interactions in which any component that fulfils the role can become involved. This is captured by a mapping $\boldsymbol{sign_{BROL}: BROL \rightarrow SIGN}$. For instance, in the Appendix, we can see that the signature of a business role is clearly identified under "interactions". For simplicity, we do not give any detail of the categorical properties of signatures in SRML-P, which are quite straightforward.

We further assume that ***SIGN*** is finitely co-complete. This means that we can compose signatures by computing colimits (amalgamated sums) of finite diagrams; typically, such diagrams are associated with the definition of complex structures of signatures, which can result from the way modules are put together as discussed in Section 4, or the way module are interconnected as discussed in Section 5.

2.3 Business protocols

Besides components, a module in SRML-P may declare a number of (external) interfaces. These provide abstractions (types) of parties that can be interconnected with

the components declared in the module either to provide or request services; this is what, in SCA, corresponds to "Entry Points" and "External Services".

External interfaces are specified through *business protocols*, the set of which we denote by ***BUSP***. Like business roles, protocols declare the interactions in which the external entities can be involved as parties; this is captured by a mapping $\boldsymbol{sign_{BUSP}: BUSP \rightarrow SIGN}$. The difference with respect to business roles is that, instead of an orchestration, a business protocol provides a set of properties that model the protocol that the co-party is expected to adhere to. In the appendix, we give as an example the business protocol that corresponds to the FlightAgent. Like for business roles, the signature of a business protocol in SRML-P is clearly identified under "interactions".

Business protocols, which model what in SCA corresponds to "external services", specify the conversations that the module expects relative to each party. Those that model what in SCA corresponds to an "entry point", specify constraints on the interactions that the module supports as a service provider. Examples of such constraints are the order in which the service expects invocations or deadlines for the user to commit, but also properties that the client may expect such as pledges on given parameters of the delivered service. It is the responsibility of the client to adhere to these protocols, meaning that the provider may not be ready to engage in interactions that are not according to the specified constraints.

2.4 Interaction protocols

Service components and external interfaces are connected to each other within modules through *internal wires* that bind the interactions that both parties declare to support and coordinate them according to a given *interaction protocol.* Typically, an interaction protocol may include routing events and transforming data provided by a sender to the format expected by a receiver. The examples given in the appendix are quite simple: they consist of straight synchronisations at the ports.

Just like business roles and protocols, an interaction protocol is specified in terms of a number of interactions. However, interaction protocols are somewhat more complex. On the one hand, an interaction protocol declares two disjoint sets of interactions; in SRML-P, this is done under the headings ROLE A and ROLE B. On the other hand, the properties of the protocol – what we called the *interaction glue* – are declared in a language defined over the union of the two roles, what we call its signature. We consider that we have a set ***IGLU*** of specifications of interaction glues together with a map $\boldsymbol{sign_{IGLU}: IGLU \rightarrow SIGN}$.

In order to model the composition of modules, we also need a way of composing interaction protocols. For that purpose, we assume that ***IGLU*** is itself a co-complete category whose morphisms $\sigma: G_1 \rightarrow G_2$ capture the way G_1 is a sub-protocol of G_2, again up to a possible renaming of the interactions and corresponding parameters. That is, σ identifies the glue that, within G_2, captures the way G_1 coordinates the interactions $\boldsymbol{sign}(G_1)$ as a part of $\boldsymbol{sign}(G_2)$. More precisely, we assume that $\boldsymbol{sign_{IGLU}}$ is a functor that preserves colimits, i.e. that the signature of a composition of protocols is a composition of their signatures.

2.5 Summary

The relationships between all these different formal domains are summarised in **Figure 2** (categories are represented with a thick line). For simplicity, we use ***sign*** as an abbreviated notation for $\boldsymbol{sign}_{BROL}$, $\boldsymbol{sign}_{BUSP}$ and $\boldsymbol{sign}_{IGLU}$.

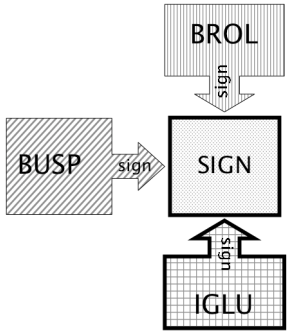


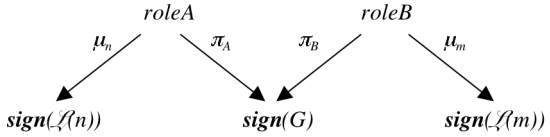
Figure 2: How the different formal domains relate to each other: ***BROL***–business roles; ***BUSP***–business protocols; ***IGLU***–interaction glue of protocols; ***SIGN***–signatures

3 Defining modules

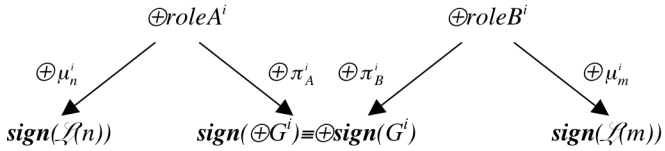
As already mentioned, modules are the basic units of composition. They include external interfaces for required and provided services, and a number of components whose orchestrations ensure that the properties offered on the provides-interfaces are guaranteed by the connections established by the wires assuming that the services requested satisfy the properties declared on the requires-interfaces.

In our formal model, a module is defined as a graph: components and external interfaces are nodes of the graph and internal wires are edges that connect them. This graph is labelled by a function $\mathcal{L}$: components are labelled with business roles, external interfaces with business protocols, and wires with connectors that include the specification of interaction protocols. An example of the syntax that we use in SRML-P for defining the graph and labelling function can be found in the Appendix.

Because a wire interconnects two nodes of the module (graph), we need some means of relating the interaction protocols used by the wire with the specifications (business roles or protocols) that label the nodes. The connection for a given node n and interaction protocol P is characterised by a morphism μ_n that connects one of the roles (A or B) of P and the signature $\boldsymbol{sign}(\mathcal{L}(n))$ associated with the node. We call a *connector* for a wire $n \overset{w}{\longleftrightarrow} m$ a structure $\langle \mu_n, \pi_n, G, \pi_m, \mu_m \rangle$:



In SRML-P, connections are defined in a tabular form that should be self-explanatory as illustrated in the Appendix. Some wires may be labelled by more than one connector. In such cases, we may compose the connectors by taking the sum of their protocols. More concretely, if we have a collection $<\mu_n^i, \pi_n^i, G^i, \pi_m^i, \mu_m^i>$ of connectors labelling a wire $n \leftrightarrow m$, we can represent it by the connector $<\oplus\mu_n^i, \oplus\pi_n^i, \oplus P^i, \oplus\pi_m^i, \oplus\mu_m^i>$ given by the diagram:



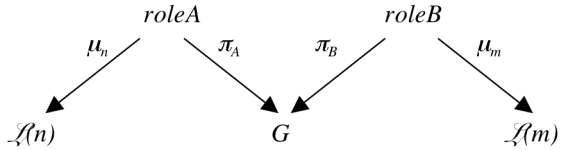
The morphisms are given uniquely by the properties of sums in ***SIGN*** [2]. This corresponds to looking at the set of connectors that labels a wire as defining a single connector. Working with wires labelled by a single connector makes it easier to define and manipulate modules.

Formally, we take a module M to consist of:

- A graph, i.e. a set *nodes(M)* and a set *wires(M)* of pairs $n \leftrightarrow m$ of nodes (elements of *nodes(M)*).
- A distinguished subset of nodes $requires(M) \subseteq nodes(M)$.
- At most one distinguished node $provides(M) \in nodes(M) \setminus requires(M)$.
- A labelling function $\mathcal{L}$ such that:
  - $\mathcal{L}(provides(M)) \in \mathbf{BUSP}$ if *provides(M)* is defined
  - $\mathcal{L}(n) \in \mathbf{BUSP}$ for every $n \in requires(M)$
  - $\mathcal{L}(n) \in \mathbf{BROL}$ for every other node $n \in nodes(M)$
  - $\mathcal{L}(n \leftrightarrow m)$ is a connector $<\mu_n, \pi_n, G, \pi_m, \mu_m>$.

A module M for which *provides(M)* is not defined corresponds to applications that do not offer any services but still require external services to fulfil their goals. They can be seen to be "agents" that, when bound to the external services that they require, execute autonomously in a given configuration as discussed below. Modules that do provide a service and can be discovered are called *service modules*.

We can expand every wire $n \leftrightarrow m$ into the following labelled directed graph:



That is, we make explicit the protocol and the connections. We denote by *expanded(M)* the result of expanding all wires in this way. Therefore, in *expanded(M)* we have the nodes of M with the same labels – business roles and protocols – and, for each wire, an additional node labelled with a protocol, two additional nodes (ports) labelled with the roles of the protocol, and directed edges from the ports labelled with signature morphisms. For instance, the expanded graph of the module depicted in **Figure 1** has the following structure:

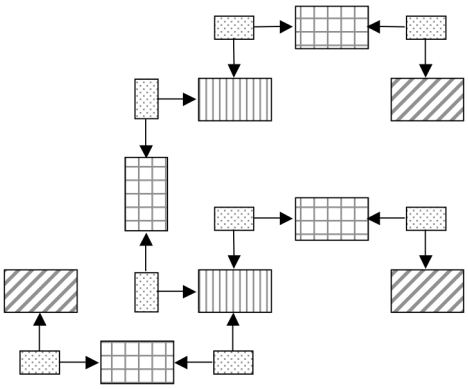


Figure 3: The expanded graph of a module; – business role; – business protocol; – interaction glue; – signature (role)

4 Semantic correctness

Section 3 defines some criteria that ensure the syntactic correctness of modules, namely the fact that the endpoints of the connectors in the wires match the labels of the nodes linked by the wire. In this section, we are concerned by the semantic correctness of service modules, i.e. the fact that the properties offered in the provides-interface are ensured by the orchestration of the components and the properties required of the other external interfaces.

The correctness condition is expressed in terms of logical entailment of properties of business protocols. The mechanisms that we provide for putting together, interconnecting and composing modules is largely independent of this logic. The particular choice of operators, their semantics and proof-theory are essential for supporting the modelling of service-based applications, i.e. for the pragmatics of "in-the-small" issues, but not for the semantics and pragmatics of modules as units of composition, i.e. for the "in-the-large" issues. 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. In SRML-P, we use Branching Time Logic [6] defined over an alphabet of events such that every interaction declared in a signature gives rise to the following set of events (see [3] for additional explanations):

`interaction`🔔	The event of initiating *interaction*.
`interaction`✉	The reply-event of *interaction*.
`interaction`✓	The commit-event of *interaction*.
`interaction`✖	The cancel-event of *interaction*.
`interaction`💣	The deadline-event of *interaction*.
`interaction`✞	The revoke-event of *interaction*.

As a consequence, we assume that we have available an entailment system (or π-institution) [2, 5] $<$***SIGN***,***gram***, $\vdash>$ where ***gram:SIGN→SET*** is the grammar functor that, for every signature Q, generates the language used for describing properties of the interactions in Q. Notice that, given a signature morphism $\sigma:Q\rightarrow Q'$, ***gram***(σ) 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 $<Q,S>$ 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 entailment. We denote by ***sign*** the forgetful functor that projects theories on the underlying signatures.

Given a specification $SP=<Q,S>$ and sets P and R_i of sentences over Q, we also use the notation

$$P \,/\!\!/\, SP \Big| \begin{matrix} R_1 \\ \vdots \\ R_N \end{matrix}$$

to indicate that $R_1 \cup \ldots \cup R_N \cup S \vdash_Q p$ for every $p \in P$, i.e. that the properties expressed by P are guaranteed by SP relying on the fact that the properties expressed in R_i hold.

As discussed in Section 2, the specifications of business roles, business protocols and interaction protocols carry a semantic meaning. We take this meaning to be defined by mappings $\boldsymbol{spec_{BROL}}$***:BROL→SPEC***, $\boldsymbol{spec_{BUSP}}$***:BUSP→SPEC*** and $\boldsymbol{spec_{IGLU}}$***: IGLU→SPEC*** that, when composed with ***sign:SPEC→SIGN***, give us the syntactic mappings discussed in Section 2.

In the case of business roles, this assumes that we can abstract properties from orchestrations, which corresponds to defining an axiomatic semantics of the orchestration language. In SRML-P, this means a straightforward translation of event-condition-action rules into Temporal Logic.

In the case of business and interaction protocols, this mapping is more of a translation from the language of external specifications to a logic in which one can reason about the properties of interactions as well as that of orchestrations. In SRML-P, the operators used in the examples given in the Appendix are translated as follows:

`a` **`before`** `b`	If *b* holds then *a* must have been true.	**AG**(b ⊃ **P**a)
`b` **`exceptif`** `a`	*b* can occur iff *b* and *a* have never occurred.	**AG**(¬**P**a∧**H**(¬b) ≡ **E**b)
`a` **`enables`** `b`	*b* can occur iff *a* has already occurred but not *b*.	**AG**(**P**a∧**H**(¬b) ≡ **E**b)
`a` **`ensures`** `b`	*b* will occur after *a* occurs, but *b* cannot occur without *a* having occurred.	**AG**(b⊃**P**a ∧ a⊃**F**b)

We further assume that the mapping $\boldsymbol{spec_{IGLU}}$ is in fact a functor, i.e. that the composition of interaction protocols preserves properties. This leads to the following extension of **Figure 2**:

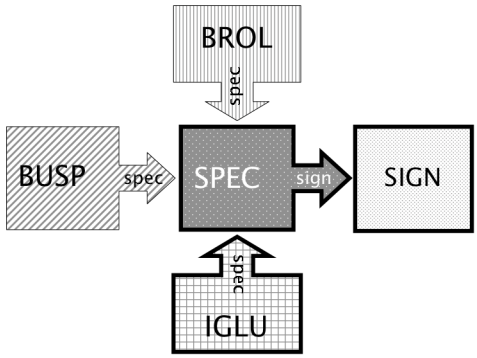


Figure 4: Relating the specification domain with the other formal domains

The correctness property of service modules relies on the fact that the orchestrations of the business roles and the properties of the interaction protocols guarantee that the properties of the requires-interfaces entail those ensured by the provides-interfaces. To express it, we need a means of referring to the fragment of the module that is concerned with components and wires, what we call the *body* of the module. Formally, we define *body(M)* for a module *M* as being the diagram of specifications and signatures that is obtained from *expanded(M)* by applying the mappings ***spec*** to all the labels (business roles, business protocols and interaction protocols). That is, we obtain the same graph as that of *expanded(M)* except that we label the nodes with the specifications of the business roles and interaction protocols, and the signatures of the business protocols. For instance, the following picture corresponds the body-diagram of the expanded-graph of **Figure 3**:

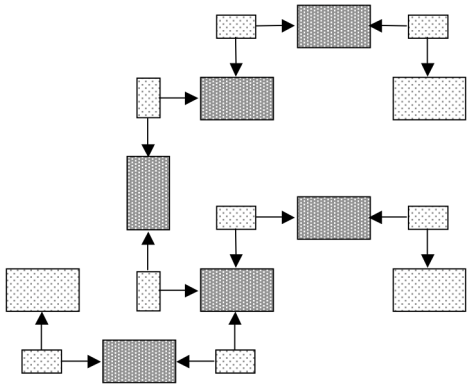


Figure 5: The body diagram of a module

We assume that the category ***SPEC*** is finitely co-complete and coordinated over ***SIGN***, which allows us to calculate the colimit (amalgamated sum) of this diagram. The colimit returns a specification ***Body****(M)* and a morphism q_n:***sign***$(\wp(n))\rightarrow$***sign****(M)* for every node *n* of *expanded(M)*.

It is helpful to detail the construction of ***Body****(M)*. Its signature ***sign****(M)* is the colimit (amalgamation) of the diagram of signatures defined by *body(M)*. This signa-

ture contains all the interactions that are involved in the module; the morphisms q_n record in which nodes each interaction is used. The set of axioms of ***Body**(M)* consists of the union of the following sets:

- For every node n labelled by a business role W, the translation $\boldsymbol{gram}(q_n)(S_W)$ where $\boldsymbol{spec}(W)=<\boldsymbol{sign}(W),S_W>$, i.e. we take the translations of the axioms of ***spec**(W)*.
- For every node n labelled by an interaction protocol P, the translation $\boldsymbol{gram}(q_n)(S_P)$ where $\boldsymbol{spec}(P)=<\boldsymbol{sign}(P),S_P>$, i.e. we take the translations of the axioms of ***spec**(P)*.

Notice that the business protocols of the external interfaces are not used for calculating ***Body**(M)*: only their signatures are used. However, because their signatures are also involved, we can operate the same kind of translation on every external interface by using the corresponding signature morphism q:

- We denote by ***Prov**(M)* the translation of the specification of the business protocol of *provides(M),* i.e. of the provide-interface of *M*.
- We denote by $\boldsymbol{Reqs}_{1..N}(M)$ the translations of the specifications of the business protocols in *requires(M),* i.e. of the requires-interfaces of *M*.

Given that all these sets of sentences are now in the language of ***sign**(M)*, the correctness property of a service module *M* can be expressed by:

$$Prov(M)\frac{}{Body(M)}\left|\begin{matrix}Reqs_1(M)\\ \vdots \\ Reqs_N(M)\end{matrix}\right.$$

That is, every property offered in the business protocol of the provides-interface must be entailed by the body of the module using the properties required in the business protocols of the requires-interfaces.

5 Composing modules

In this section, we discuss the mechanisms through which modules can be assembled to create systems and modules can be created from systems. These mechanisms are similar to those provided in SCA, i.e. they provide a means of linking requires-external interfaces of a module with provides-external interfaces of other modules. In SRML-P, we provide only abstract models of such links, which we call *external wires*. That is, we remain independent of the technologies through which interfaces are bound to parties, which depend on the nature of the parties involved (BPEL processes, Java programs, databases, *inter alia*). In summary, external wires carry a proof-obligation to ensure that the properties offered by the provides-interface are implied by those declared in the requires-interfaces.

A system is a directed acyclic graph in which nodes are labelled by modules and edges are labelled with so-called “bindings” or “external wires”. A binding for an edge $n \rightarrow k$ between modules M_n and M_k consists of:

- A node $r \in requires(M_n)$, i.e. one of the requires-interfaces of M_n. This node cannot be used by any other binding. Let this node be labelled with S_r.
- A specification morphism $\rho: \mathbf{spec}(S_r) \rightarrow \mathbf{spec}(S_p)$ where S_p is the business protocol of $provides(M_k)$, i.e. of the provides-interface of M_k.

In other words, bindings connect a requires-interface of one module to the provides-interface of another module such that the properties of the requires-interface are implied by the properties of the provides-interface.

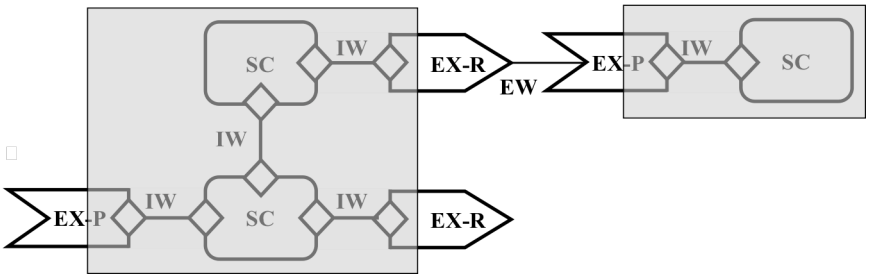


Figure 6: An assembly of modules defining a SRML-P system; EW–external wire

SRML-P also supports a way of offering a system as a module, i.e. of turning an assembly of services into a composite service that can be published and discovered on its own. This can be useful, for instance, when one wants to put together a number of services that, individually, offer only partial matches for a given required external interface but, in a suitable configuration, can provide a suitable match. The operation that collapses a system into a module internalises the external wires and forgets the external specifications.

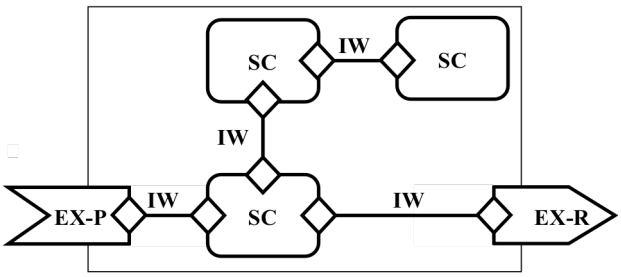
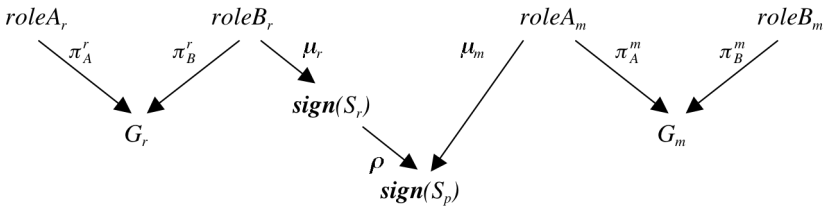


Figure 7: The previous system turned into a module

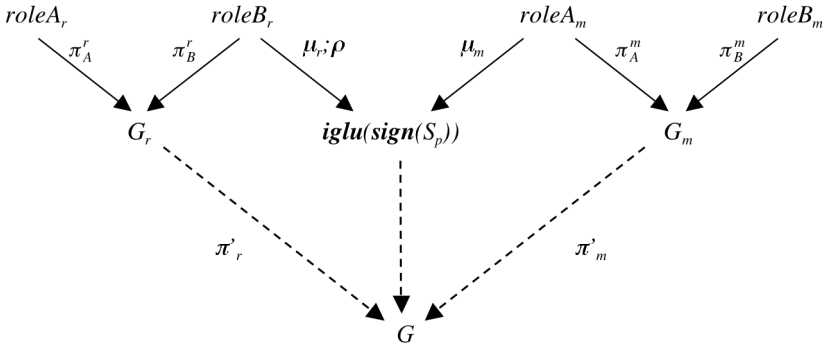
Formally, a module may be created from every (finite) weakly connected system by internalising the bindings. The resulting module M is as follows:

- The graph of M is obtained from the sum (disjoint union) of the graphs of all modules involved in the system by eliminating, for every edge $n \rightarrow k$ of the system, the nodes r (requires) of M_n and $provides(M_k)$, and adding, for every such edge $n \rightarrow k$ of the system, an edge $i \leftrightarrow j$ between any two nodes i and j such that $i \leftrightarrow r$ is an edge of M_n and $provides(M_k) \leftrightarrow j$ is an edge of M_k.
- The labels are inherited from the graphs of the modules involved, except for the new edges $i \leftrightarrow j$. These are calculated by merging the connectors that la-

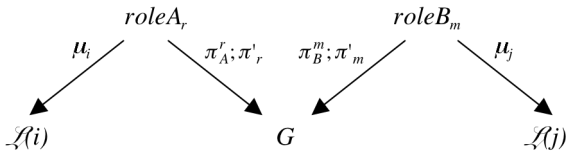
bel $i \leftrightarrow r$ and $provides(M_k) \leftrightarrow j$. The interaction protocol of the new connector is obtained through the colimit diagram below where $m=provides(M_k)$.



This composition is defined by the following colimit diagram in ***IGLU***:



The rest of the connector is defined by the morphisms μ_i of $i \leftrightarrow r$ and μ_j of $provides(M_k) \leftrightarrow j$:



- *requires(M)* consists of the remaining requires-interfaces.
- *provides(M)* consists of the remaining provides-interface, if one remains. Notice that the connectivity of the graph implies that at most one provides-interface can remain.

The colimits calculated in order to obtain the protocol of the new connectors are expressed over a "diagram" that involves both signatures (those of the external interfaces and the ports) and protocols.

For this construction to make sense, we assume that the category ***IGLU*** is coordinated over ***SIGN*** [2]. This means that we have a canonical way of lifting signatures to interaction protocols that respects the interactions. In other words, every signature can be regarded as an interaction protocol through which the "diagram" above defines a diagram in ***IGLU***, thus allowing for the colimit to be computed.

The following picture depicts the graph involved in the composition considered in **Figure 6**:

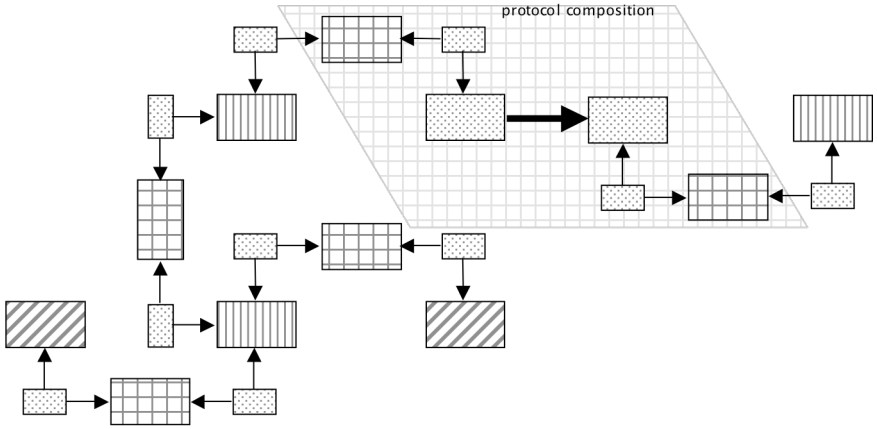


Figure 8: The graphs involved in a composition; the diagram of interaction protocols involved in the internalisation of the binding is singled out

The graph obtained from the internalisation of the binding is the one that expands the module identified in **Figure 7**:

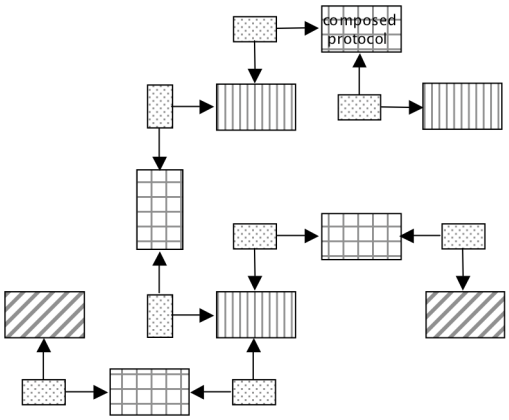


Figure 9: The expanded graph of the composite module

6 Concluding Remarks and Further Work

In this paper, we have described some of the primitives that are being proposed for the SENSORIA Reference Modelling Language in order to support building systems in service-oriented architectures using "technology agnostic" terms. More specifically, we have focused on the language that supports the underlying composition model. This is a minimalist language that follows a recent proposal for a Service Component Architecture [7] that "builds on emerging best practices of removing or abstracting middleware programming model dependencies from business logic". However, whereas the SCA-consortium concentrates on the definition of an open

specification that supports a variety of component implementation and interface types, and on the deployment, administration and configuration of SCA-based applications, our goal is to development a mathematical framework in which service-modelling primitives can be formally defined and application models can be reasoned about.

Our composition model relies on the notion of module, which we adapted from SCA. Modules can be discovered and bound to other modules at run-time to produce configurations. We proposed a formal model for module assembly and composition in line with algebraic notions of component [1]. We intend to do further research on the way SRML-P modules relate to such established notions.

We are currently developing a notion of configuration for SRML-P as a collection of components wired together that models a run-time composition of service components. A configuration results from having one or more clients using the services provided by a given module, possibly resulting from a complex system, with no external interfaces, i.e. with all required external interfaces wired-in. It is at the level of configurations that we address run-time aspects of service composition such as sessions, as well as notions of persistence.

Acknowledgments

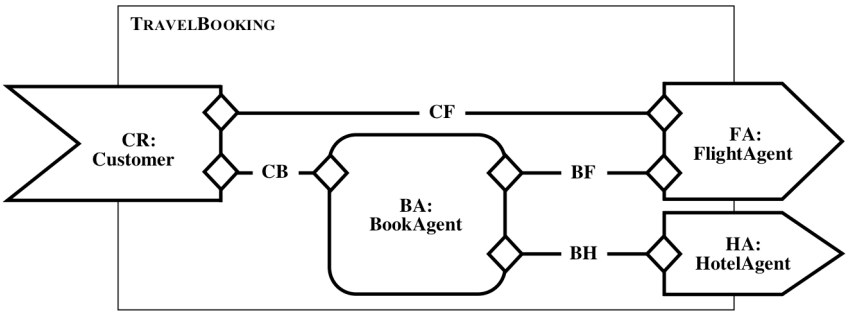
J. Fiadeiro was partially supported by a grant from the Royal Society (UK) while on study leave from the University of Leicester at the University of Pisa during April and May 2006. A. Lopes was partially supported by the Foundation for Science and Technology (Portugal) during an extended stay at the University of Pisa during May 2006.

References

1. H. Ehrig, F. Orejas, B. Braatz, M. Klein, M. Piirainen (2004) A component framework for system modeling based on high-level replacement systems. *Software Systems Modeling* 3:114–135
2. J. L. Fiadeiro (2004) *Categories for Software Engineering*. Springer, Berlin Heidelberg New York
3. J. L. Fiadeiro, A. Lopes, L. Bocchi (2006) *The SENSORIA Reference Modelling Language: Primitives for Service Description and Composition*. Available from www.sensoria-ist.eu
4. J. L. Fiadeiro, A. Lopes, L. Bocchi (2006) A formal approach to service-oriented architecture. In: M. Bravetti, M. Nunez, G. Zavattaro (eds) *Web Services and Formal Methods. LNCS, vol 4184*. Springer, Berlin Heidelberg New York, pp 193–213
5. J. Goguen, R. Burstall (1992) Institutions: abstract model theory for specification and programming. *Journal ACM* 39(1):95–146
6. R. Goldblatt (1987) *Logics of Time and Computation*. CSLI, Stanford
7. SCA Consortium (2005) *Building Systems using a Service Oriented Architecture*. Whitepaper available from www-128.ibm.com/developerworks/library/specification/ws-sca/

Appendix – TravelBooking

In this appendix, we provide parts of a typical travel-booking process involving a flight and a hotel agent. The module – *TRAVELBOOKING* – that defines this composite service exposes to the environment an interface for booking a flight and a hotel for a given itinerary and dates. External services are requested in order to offer the service behaviour that the module declares to provide.



TRAVELBOOKING consists of:

- CR – the external interface of the service provided by the module, of type *Customer*;
- FA – the external interface of a service required for handling the booking of flights, of type *FlightAgent*;
- HA – the external interface of a service required for handling the booking of hotels, of type *HotelAgent*;
- BA – a component that coordinates the business process, of type *BookAgent*;
- *CB*, *CB, BF*, *BH* – four internal wires that make explicit the partner relationship between *CR* and *BA*, *CR* and *FA*, *BA* and *FA*, and *BA* and *FA*.

```
MODULE TravelBooking(bval:time) is

  PROPERTY
    bval<HA.hval ∧ FA.fval>bval+HA.hresp

  COMPONENTS
    BA: BookAgent(fresp,hresp)

  PROVIDES
    CR: Customer

  REQUIRES
    FA: FlightAgent(fval)
    HA: HotelAgent(hval)
```

WIRES

BA BookingAgent		BF		FA FlightAgent
s&r bookFlight	S_1		R_1	**r&s** lockFlight
🔔 from	i_1		i_1	🔔 from
to	i_2	Straight	i_2	to
out	i_3		i_3	out
in	i_4		i_4	in
✉ fconf	o_1		o_1	✉ fconf
rcv fConfirm	R_1	Straight	S_1	**snd** flightAck
🔔 result	i_1		i_1	🔔 result

[…]

SPECIFICATIONS

BUSINESS ROLE BookAgent(fresp,hresp:time) **is**

INTERACTIONS

```
r&s bookTrip
   🔔 from,to:airport, out,in:date
   ✉ tconf:(fcode,hcode)
s&r bookFlight
   🔔 from,to:airport, out,in:date
   ✉ fconf:fcode
s&r bookHotel
   🔔 checkin:date, checkout:date
   ✉ hconf:hcode
snd tAck
   🔔 result:bool
rcv fConfirm
   🔔 result:bool
```

ORCHESTRATION

```
local s:[0..6], fconf:fcode, hconf:hcode,
      out,in:date, from,to:airport,
      frep, hrep: boolean
initialisation s=0
termination s=3 ∨ (s=6 ∧ today≥out)
transition TOrder
   triggeredBy bookTrip🔔?
   guardedBy s=0
   effects from'=bookTrip.from
      ∧ to'=bookTrip.to
      ∧ out'=bookTrip.out
      ∧ in'=booktrip.in
      ∧ out'≥today ⊃ s'=1
      ∧ out'<today ⊃ s'=3
   sends out'>today ⊃ bookFlight🔔!
         ∧ bookFlight.from=from'
         ∧ bookFlight.to=to'
         ∧ bookFlight.out=out'
         ∧ bookflight.in=in'
         ∧ alertDate🔔!
         ∧ alertDate.Ref="flight"
         ∧ alertDate.Interval=fresp
      ∧ out'≤today ⊃ bookTrip✉!
         ∧ bookTrip.Reply=false
```

State variables for storing data that may be needed during the orchestration.

s is used for control flow, i.e. for encoding an internal state machine.

The other state variables are used for storing data transmitted through the parameters of interactions.

Property guaranteed for the initial state.

Property that determines when the orchestration has terminated.

If *var* is a state variable, *var'* denotes its value after the transition; this expression can be used in both "effects" and "sends".

A request to travel on a date already passed leads immediately to a final state.

today is an external services that we assume to be globally available; it provides the current date.

In response to a request for travelling in a future date, a flight request is issued and a timeout is set with the duration that the agent is willing to wait for a reply.

alertDate is also a service that is globally available; it replies when the duration set-up in the parameter *Interval* elapses. We use the parameter *Ref* to correlate different alerts that are sent.

BUSINESS PROTOCOL FlightAgent(fval:time) **is**

INTERACTIONS

r&s lockFlight
🔔 from,to:airport, out,in:date
✉ fconf:fcode

s&r payRequest
🔔 amount:nat, benef:account,
bank:servRef
✉ pay:payData

snd payAck
🔔 result:bool

snd payRefund
🔔 amount:nat

BEHAVIOUR (fref:string)

lockFlight🔔? **exceptif** true

lockFlight✉! ∧ lockFlight.Reply
⊃ alertDate🔔!
∧ alertDate.Ref=fref
∧ alertDate.Interval≥fval

lockFlight💣! ⊃ alertDate✉?
∧ alertDate.Ref=fref

lockFlight✓? **ensures** payRequest🔔!

payRequest✉? ∧ payRequest.Reply
ensures payAck!

today<lockFlight.out ⊃ (payAck!
∧ payAck.result **enables** lockFlight✞?)

lockFlight✞? **ensures** payRefund🔔!
∧ payRefund.amount
>payRequest.amount*0.9

In the initial state, FA is required to be ready to receive a request for a flight.

The timeout for flight reservations is at least fval.

FA is required to request the payment after receiving the commit.

FA is required to send *payAck* to acknowledge the reception of a successful payment.

FA is required to accept the revoke of a flight booking until the day of departure and provide a refund of at least 90% of its cost.

INTERACTION PROTOCOL Straight.I(d_1,d_2,d_3,d_4)O(d_5) **is**

ROLE A

s&r S_1
🔔 i_1:d_1, i_2:d_2, i_3:d_3, i_4:d_4
✉ o_1:d_4

ROLE B

r&s R_1
🔔 i_1:d_1, i_2:d_2, i_3:d_3, i_4:d_4
✉ o_1:d_4

COORDINATION

$S_1 \equiv R_1$
$S_1.i_1=R_1.i_1$
$S_1.i_2=R_1.i_2$
$S_1.i_3=R_1.i_3$
$S_1.i_4=R_1.i_4$
$S_1.o_1=R_1.o_1$