Specifying and Composing Interaction Protocols for Service-Oriented System Modelling*

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

[1] Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
{abreu,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 and discuss a formal, high-level approach to the specification and composition of interaction protocols for service-oriented systems. This work is being developed within the SENSORIA project as part of a language and formal framework supporting the modelling of complex services at the business level, i.e. independent of the underlying platform and the languages in which services are programmed and deployed. Our approach is based on a novel language and logic of interactions, and a mathematical semantics of composition based on graphs. We illustrate our approach using a case study provided by Telecom Italia, one of our industrial partners in the project.

1 Introduction

SENSORIA – an IST-FET Integrated Project on *Software Engineering for Service-Oriented Overlay Computers* – is defining a formal framework for modelling service-oriented systems in a broad sense that encompasses and generalises the methods and techniques that are either available or envisioned for Web Services [1], as well as other platforms such as Grid Computing [9]. One of the strands of the project is the definition of a reference modelling language – SRML – that can address the higher levels of abstraction of "business modelling" by providing modelling primitives that are independent of the languages and the middleware infrastructure over which services are programmed. This includes a mathematical semantics that can support different kinds of analysis and in relation to which techniques for the deployment, publication, discovery and binding of services can be defined and proved to be correct.

In [6], we presented a preliminary account of our approach and the way it relates to the Service Component Architecture (SCA) [13], namely the notion of module that we adopt for describing complex services and support service discovery and composition. An algebraic semantics of SRML modules and module composition can be

* 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*.

J. Derrick and J. Vain (Eds.): FORTE 2007, LNCS 4574, pp. 358–373, 2007.

found in [7]. In this paper, we report in more detail on one of the key ingredients of service description and composition: the interaction protocols that are responsible for interconnecting the different parties that are involved in a composite service. The challenge here is twofold. On the one hand, to provide a formal model that is rich enough to capture the characteristics of interactions that are typical of service-oriented systems. This includes interactions that are 'conversational', i.e. that cannot be characterised by a transition involving only initial and final states. On the other hand, to make the interaction protocols independent of the way the parties involved in them engage in the interactions, for instance the workflows that determine when the parties actually interact. This is important for dynamic, run-time service discovery and binding, and also for reuse.

In Section 2, we discuss and justify the role that, in our approach, we assign to interaction protocols. In Section 3, we present the language that we use for describing and using interaction protocols in the connectors that establish wires between parties of a complex service. Finally, in Section 4, we present an algebraic semantics for interaction protocols. Throughout the paper, we use examples from a case study developed with Telecom Italia, one of our industrial partners in SENSORIA: the "Call and Pay Taxi through SMS" scenario.

2 Modelling Complex Services in SRML

From the more abstract point of view of systems modelling, i.e. once we abstract from the nature of the languages and platforms over which services are deployed, the main challenge raised by service-oriented systems is in the number of autonomic entities involved and the complexity of the interactions within them. That is, the complexity that matters is not so much in the "size" of the code through which such entities are programmed (size is a design time issue) but on the number, intricacy and dynamicity of the interactions in which they will be involved, what in [4] we have called *social complexity*.

This is why it is so important to put the notion of interaction at the centre of research in service-oriented system modelling. This is also why new methods and formal techniques become necessary. For instance, from an algebraic point of view, social complexity raises new challenges in that it does not make sense to see service-oriented systems as being compositions, in an algebraic sense, of simpler components: there is not a notion of whole to which the parts contribute but, rather, a number of autonomic entities that interact with each other through "interaction protocols" that are external to and independent from those entities.

2.1 The Module Structure

In what concerns the definition of a modelling language that can tackle these new challenges, our approach within SENSORIA is based on a notion of module through which we specify complex services and break the complexity of running systems by recognising larger chunks (sub-configurations) that have a meaning in the application domain, i.e. correspond to "business activities". This notion of module, which is inspired by recent work of *Service Component Architecture (SCA)* [13], supports the modelling of composite services as entities whose business logic involves a number of interactions among more elementary service components as well as the invocation

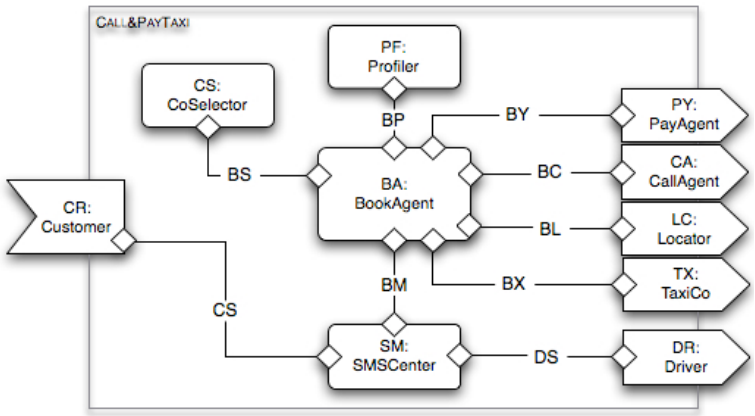
of services provided by other parties. As in SCA, interactions are supported on the basis of service interfaces defined in a way that is "independent of the hardware platform, the operating system, hosting middleware and the programming language used to implement the service".

In order to illustrate our approach, we are going to use the *Call and Pay Taxi* service scenario used by Telecom Italia, one of the partners of SENSORIA, within its R&D activities on Parlay X telecommunications web services [1]. This is a complex service that involves different telecommunication services provided by mobile networks and other external parties in order to provide users the ability to call a taxi and pay for the ride by sending SMS's to a specified number (4777 in [1]). The business process enacted by the service consists of the following steps:

- The user sends an SMS to 4777 to ask for a taxi at his/her current location.
- The service retrieves information about the user from *User Profiler*, and its location from *User Locator Service*.
- The service selects a taxi company at the user's location.
- The service uses a *Call Agent* to set up a voice call between the user and the taxi company.
- The service sends the user and taxi driver an SMS with the taxi number and a "call-code" identifying the transaction.
- After the taxi ride, and in order to authorise the payment, the user sends an SMS with the information previously received and the amount to be paid.
- The service sends a charging request to a *Payment Service*.
- The taxi driver and the user receive a notification of the outcome of the payment via another SMS.

In order to model the *Call&PayTaxi* service through a module in SRML, we need to decide which entities of the scenario description are to be represented as internal components – in the sense that they are deployed when the module is instantiated – and which correspond to parties that need to be procured externally at run-time, in which case they are modelled by what we call *external interfaces*.

The module that we propose has the following structure:



2.2 The Provides-Interface

Every service module in SRML has one distinguished external interface, what we call a *provides*-interface or EX-P for short. The EX-P declares the interactions and protocol that are supported between the service and any service requester. The EX-P of *Call&PayTaxi* is declared to be *CR* of type *Customer* – a business protocol that consists of a set of interactions and a specification of the dependencies that exist between them, including the order in which they are expected to occur. This subsumes what, in [2], are called external specifications i.e., the specification of which message exchange sequences are supported by the service, for example expressed in terms of constraints on the order in which service operations should be invoked.

This is how we specify a business protocol in SRML:

```
BUSINESS PROTOCOL Customer(myNumber:phoneNum) is
INTERACTIONS
   snd callTaxiOUT
   rcv callTaxiIN
      🔔 text:string
   snd payTaxiOUT
      🔔 text:string
   rcv payTaxiIN
      🔔 text:string
BEHAVIOUR
   initiallyEnabled callTaxiOUT🔔?
   P_callTaxiOUT🔔? ensures callTaxiIN🔔!
   P_callTaxiIN🔔! ∧ callTaxiIN.text≠'NA' enables payTaxiOUT🔔?
   P_payTaxiOUT🔔? ensures payTaxiIN🔔!
```

A business protocol declares the interactions maintained by the service under what we call an *interaction signature* (or *signature*, for short). In the example above, we use one-way asynchronous interactions that correspond to the SMS's sent (*OUT*) and received (*IN*) by the customer. Notice that there is no declaration of which components inside the service are co-parties in these interactions; co-parties are identified through wires as discussed below, which also specify the protocol that coordinates the interaction between the two parties.

One-way interactions may have parameters, which are declared under 🔔. In the example above, these correspond to the text of the SMS. The business protocol itself has a parameter: *myNum* of type *phoneNum*. This parameter is instantiated with the phone number of the customer when the actual customer is bound to the *Call&PayTaxi* service.

Further to a signature, a business specification includes the properties of the conversation that any customer can have with the service. The first property declares that, initially (i.e. when the service is bound to the customer), the co-party is ready to accept a call for *callTaxiOUT*. The second property declares that the fact that the co-party has received a call for *callTaxiOUT* ensures that the service will issue a *callTaxiIN*. The third property declares that, if the *callTaxiIN* has been issued with a text other than 'No taxi available', the service is ready to receive a payment *payTaxiOUT*. Finally, the fourth property ensures that, having received a *payTaxiOUT*, the service will issue an

acknowledgment *payTaxiIN*. The language in which these properties are expressed uses abbreviations of a temporal logic that we briefly discuss in Section 3.

2.3 Requires-Interfaces

The service provided through *CR* results from a business process that involves a number of internal components that may need to invoke external services specified in the module through what we call *requires*-interfaces (EX-R's for short). The discovery process for any given EX-R takes place at run-time when given declared triggers occur, and returns a service that implements a module whose EX-P matches the EX-R. Through the binding mechanisms of the underlying middleware, the components through which the discovered service is implemented become connected to those of the client service through the interaction protocols specified in the wires. The system thus assembled executes according to the orchestration that results from the assembly.

The external parties defined in our example are:

- The user locator *LC*.
- The call agent *CA* responsible for establishing phone calls.
- The payment agent *PY*.
- The taxi driver *DR*.
- The taxi company *TX*.

The specification of an EX-R is given by a business protocol much in the same way as for the provides-interface. As an example, consider the conversation with the taxi company *TX*:

```
BUSINESS PROTOCOL TaxiCo is
  INTERACTIONS
    r&s contactCompany
      🔔 userNum:phoneNum, language:lang
      ✉ taxiNum:reference, callCode:reference,
        driverNum:phoneNum
    snd requestCall
      🔔 operatorNum
  BEHAVIOUR
    initiallyEnabled contactCompany🔔?
    P_contactCompany🔔? ensures requestCall🔔!
```

We use a two-way interaction – *contactCompany* of type **r&s** – which means that the taxi company is required to be able to engage in an interaction that is initiated by the co-party and issues a reply. The parameters of the reply event are declared under ✉; in our case, they consist of the taxi number, a code, and the phone number of the driver. The signature of this business protocol also includes a one-way interaction of type **snd**: the taxi company is required to request a phone call with the customer.

The properties required of the taxi company are as follows: when bound to the module, this external service should be ready to accept the event *contactCompany*🔔*?*, after which it is required to issue a *requestCall*.

2.4 Service Components

A component in SRML corresponds to a resource that is used internally in the sense that it is not visible to whatever client becomes bound through the EX-P. Such resources are tightly bound inside the implementations of the module; they can be web-services, Java components, interfaces to databases, legacy systems, and so on.

The internal components that we decided to include are:

- A user profiler *PF*, which can be seen to correspond to a database of users owned and managed by the company providing the *Call&PayTaxi* service.
- The SMS centre *SM*, which is made available via a fixed phone number – 4777 in the case at hand.
- A component *BA* of type *BookAgent* that is responsible for orchestrating the interactions between all the elements of the module.
- The company selector *CS* that is used by *BA* to choose the most suitable taxi company for a given location and language.

Notice that, in SRML, the orchestration of the module is not necessarily delegated to a single internal component. The overall workflow of the business process emerges from the interconnections between the components of the module as captured through the interaction protocols of the wires that connect them.

Service components are specified through what we call *business roles*. These include a signature as for business protocols but, instead of a set of properties, we specify a transition system that captures the execution pattern of the component; we refer to this pattern as the *orchestration* of the component. For instance, consider the business role that models the SMS centre:

```
BUSINESS ROLE SMSCentre(serviceNum:phoneNum) is
  INTERACTIONS
    snd sendSMS[k:int]
        🔔 origin:phoneNum, destination:phoneNum, text:string
    rcv receiveSMS[k:int]
        🔔 origin:phoneNum, destination:phoneNum, text:string
    snd forwardIN[k:int]
        🔔 origin:phoneNum, text:string
    rcv forwardOUT[k:int]
        🔔 destination:phoneNum, text:string
  ORCHESTRATION
    transition inForward
      triggeredBy receiveSMS[i]🔔?
      guardedBy receiveSMS[i].destination=serviceNum
      sends forwardIN[i]🔔!
        ∧ forwardIN[i].origin=receiveSMS[i].origin
        ∧ forwardIN[i].text=receiveSMS[i].text
    transition outForward
      triggeredBy forwardOUT[i]🔔?
      sends sendSMS[i]🔔!
        ∧ sendSMS[i].origin=forwardOUT[i].origin
        ∧ sendSMS[i].destination=serviceNum
        ∧ sendSMS[i].text=forwardOUT[i].text
```

In this example, interactions have *key*-parameters in addition to the normal ones. This allows us to handle occurrences of multiple interactions of the same type; in this case, sending and receiving SMS's. The wires that connect the SMS centre to other parties are responsible for deciding which key parameter is used for handling the relevant interactions. This is discussed in Section 3.

The business role has itself a parameter – *serviceNum* of type *phoneNum*. The idea is to define not one but a family of business roles, each modelling a component that operates a particular SMS service. Because SMS centres handle interactions in a way that is independent of the service number, it makes sense to parameterise their specification. Such parameters are fixed when we need a specific business role in a module; for instance, in *Call&PayTaxi*, we declare *SM:SMSCentre(4777)*, i.e. the component *SM* is of type *SMSCentre(4777)*.

Notice that no relative ordering is specified on the transitions; the orchestration of business roles can be much more complex, precisely to capture the richness of workflows that arise in business modelling [6].

3 The Role of Interaction Protocols in SRML

As mentioned several times in the previous section, we rely on what we call *wires* to establish and coordinate interactions between parties. More concretely, we have seen how components and external parties are modelled without any direct reference to the co-parties involved in the interactions. This is because, on the one hand, we want the interconnections between components and external parties to be established at run-time as a result of service discovery and binding and, on the other hand, we want to promote reuse at design time. Therefore, we treat all names as being local and rely on explicit name bindings to establish which are the peers involved in each interaction.

3.1 The Logic of Interactions

Before explaining how wires are specified in SRML, it is important to make a few remarks about the logic that is being developed for interactions. Our logic is based on μ*UCTL*, a formalism being developed within SENSORIA for qualitative analysis [11]. This formalism is based on doubly-labelled transition systems which consist of:

- a set Q of states;
- an initial state q_0;
- a set *Act* of observable events;
- a transition relation $q \xrightarrow{\alpha} q'$ where α is a subset of $Act! \cup Act?$ with $Act!=\{e! \mid e \in Act\}$ and $Act?=\{e? \mid e \in Act\}$;
- a labelling function assigning to every atomic proposition p the set of states in which p is true.

By *e!* we denote the action of the initiating party sending the event *e* and by *e?* the action of its co-party processing it. In SRML, the set *Act* has more structure in that the events are generated from asynchronous interactions according to their type as shown in the figure below. We also allow synchronous interactions but, for simplicity, we do not discuss them in the paper. See [6] instead.

Interactions involve two parties and can be in both directions, i.e. they can be conversational. Interactions are described from the point of view of the party in which they are declared, i.e. "receive" means invocations received by the party and sent by the co-party, and "send" means invocations made by the party. We distinguish several events that can occur during such interactions:

`interaction`🔔	The event of initiating *interaction*
`interaction`✉	The reply-event of *interaction* (**`r&s`** and **`s&r`** only)
`interaction`✓	The commit-event of *interaction* (**`r&s`** and **`s&r`** only)
`interaction`×	The cancel-event of *interaction* (**`r&s`** and **`s&r`** only)
`interaction`✞	The revoke-event of *interaction* (**`r&s`** and **`s&r`** only)

The reply, commit, cancel and revoke events capture the conversational aspects of interactions. They are discussed in more detail in [6] together with the handling of deadlines, pledges and compensations. Being asynchronous, interactions do not require the party that initiates an event to block until the co-party receives it. As discussed in the next sub-section, there is a delay between sending and receiving an event that depends on the wire that connects the two parties. Notice that by *e?* we do not denote the act of *receiving* but of *processing* the event. This is because the co-party may not be in a state in which it can process the event *e*; if that is the case, *e!* occurs but *e?* does not. For instance, in the orchestration of the SMS centre we specified that events *receiveSMS[i]*🔔 are only processed when their destination is the number of the SMS service.

Because interactions are asynchronous, the sender never blocks; however, there is no guarantee that the co-party will process an event. This is why it is important to state in the business protocols when the co-party is ready to process the events initiated by the party. For instance, in *Customer* we declared that the service is ready to process *callTaxiOUT*🔔, and that it is ready to process *payTaxiOUT*🔔 after sending *callTaxiIN*🔔 with a positive reply. If the customer calls these events in other circumstances, there is not guarantee that the service will process them.

The logic $\mu UCTL$ uses the typical minimal fixed point operator based on a strong next operator [11]. In support of modelling, we tend to use abbreviations, as illustrated in the business protocols of Section 2, which can be defined as in [6].

3.2 Connectors

Wires bind the names of the interactions and specify the protocols that coordinate the interactions between two parties. For instance, this is how we declare the wire *CS* that connects the customer *CR* and the SMS centre *SM*:

WIRES

CR Customer(my)		**CS**		**SM** SMSCentre(4777)
snd callTaxiOUT	S_1	**SendEmptySMS** (my,4777)	R_1 i_1 i_2 i_3	**rcv** receiveSMS[1] 🔔 origin destination text
rcv callTaxiIN 🔔 text	R_1 i_1	**SendSMS** (my,4777)	S_1 i_1 i_2 i_3	**snd** sendSMS[1] 🔔 origin destination text
snd payTaxiOUT 🔔 text	S_1 i_1	**SendSMS** (my,4777)	R_1 i_1 i_2 i_3	**rcv** receiveSMS[2] 🔔 origin destination text
rcv payTaxiIN 🔔 text	R_1 i_1	**SendSMS** (my,4777)	S_1 i_1 i_2 i_3	**snd** sendSMS[2] 🔔 origin destination text

Every wire is composed of one or more *connectors* each of which corresponds to a row of the table above. In SRML, connectors are specified independently of each other so as to increase reusability at design time. Every connector consists of an interaction protocol and two bindings. As an example, consider the connector:

CR Customer(my)		**CS**		**SM** SMSCentre(4777)
snd callTaxiOUT	S_1	**SendEmptySMS** (my,4777)	R_1 i_1 i_2 i_3	**rcv** receiveSMS[1] 🔔 origin destination text

The interaction protocol of this connector is specified as follows:

INTERACTION PROTOCOL SendEmptySMS(cn,sn:phoneNum) **is**

ROLE A

snd S_1

ROLE B

rcv R_1

🔔 i_1:phoneNum
i_2:phoneNum
i_3:string

COORDINATION

$R_1 \equiv S_1$
$R_1.i_1$=cn
$R_1.i_2$=sn
$R_1.i_3$=''

Just like business roles and protocols, an interaction protocol is specified in terms of a number of interactions. Because interaction protocols establish a relationship between two parties, the interactions in which they are involved are divided in two subsets called *roles* – A and B. The "semantics" of the protocol is provided through a

collection of properties – what we call the *interaction glue* – that establish how the interactions are coordinated. This may include routing events and transforming sent data to the format expected by the receiver.

For instance, in the example above, the roles are quite simple: each consists of a single interaction. The properties established by the glue are as follows:

- The first declares that the interactions declared in both roles are identical, i.e. that their corresponding events are the same. More precisely, this is an abbreviation for $R_1!\equiv S_1! \wedge R_1?\equiv S_1?$.
- The other three properties identify the parameters of the interaction of role B: they are all fixed by the parameters of the protocol and the fact that the text message is empty.

In addition, every wire *W* has an attribute *W.delay* that determines the maximum delay that can take place in the transmission of events between the parties, i.e. between sending and receiving.

The interaction protocol used in the remaining connectors is quite straightforward:

```
INTERACTION PROTOCOL SendSMS(cn,sn:phoneNum) is
   ROLE A
      snd S1
         🔔 i1:string
   ROLE B
      rcv R1
         🔔 i1:phoneNum
            i2:phoneNum
            i3:string
   COORDINATION
      R1 ≡ S1
      R1.i1=cn
      R1.i2=sn
      R1.i3=S1.i1
```

That is, the protocol just copies the text of the message.

In a connector, the interaction protocol is bound to the parties via mappings from its roles to the signatures of the parties, which is indicated in the rows of the table. The advantage of separating the definition of the interaction protocols from their use in the wires is that it promotes reuse.

As another example, consider the following connectors that are part of the wire that connects the booking agent *BA* and the SMS centre *SM*:

BA BookAgent	◇—	**BM**	—◇	**SM** SMSCentre(4777)
snd informCustomer 🔔 driverPhone taxiNum callCode location	S_1 i_1 i_2 i_3 i_4	**Internal2SMS**	R_1 i_1 i_2	**rcv** forwardOUT[1] 🔔 destination text
rcv payTaxi 🔔 amount taxiNum callCode	R_2 i_1 i_2 i_3	**SMS2Internal**	S_1 i_1 i_2	**snd** forwardIN[2] 🔔 origin text

The first connector concerns the SMS that the booking agent needs to send to the customer with information about the taxi. According to the business role *SMSCentre*, *forwardOUT[1]🔔?* triggers *sendsSMS[1]🔔!* which we have just seen is the event *callTaxiIN🔔!* of the customer *CR*. The corresponding business protocol needs to convert the data received from *BA* into a text message that can then be sent to *CR*:

```
INTERACTION PROTOCOL Internal2SMS is
  ROLE A
    snd S1
      🔔 i1:phoneNum
         i2:reference
         i3:string
         i4:geoData
  ROLE B
    rcv R1
      🔔 i1:phoneNum
         i2:string
  LOCAL
    textify:reference,string,geoData→string
  COORDINATION
    S1 ≡ R1
    S1.i1=R1.i1
    R1.i2=textify(S1.i2,S1.i3,S1.i4)
```

The conversion is performed by an operation *textify* that is internal to the interaction protocol in the sense that the implementation of the interaction protocol needs to provide a method call to an object that can perform the operation.

The other connector performs a dual operation: it forwards the SMS received from the customer via *payTaxiIN🔔!* to the booking agent, for which it needs to parse the text message received from *CR*:

```
INTERACTION PROTOCOL SMS2Internal is
  ROLE A
    snd S1
      🔔 i1:phoneNum
         i2:text
  ROLE B
    rcv R1
      🔔 i1:moneyValue
         i2:reference
         i3:string
  LOCAL
    parseMV:string→moneyValue
    parseRF:string→reference
    parseST:string→string
  COORDINATION
    S1 ≡ R1
    R1.i1=parseMV(S1.i2)
    R1.i2=parseRF(S1.i2)
    R1.i3=parseSR(S1.i2)
```

All these examples specify very simple interaction protocols but the formalism is expressive enough to handle more complex connectors, especially through the use of

state variables. This is particularly relevant when we are reusing existing component to define the module and we need to interconnect them without changing their code.

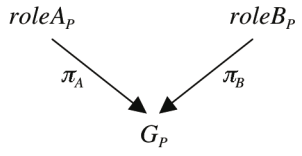
3.3 Algebraic Semantics of Connectors

An algebraic formalisation of this notion of module and module composition has been given in [7] from the point of view of a notion of correctness defined based on the theory of institutions [12]. In this section, we explore the algebraic structure of connectors in more detail and in a more general setting that does not require the level of detail that we used in [7].

As motivated in Section 2, interactions constitute the core and the unifying element of the proposed approach to systems modelling: all the models that we work with – business roles, business protocols and interaction protocols – are based on structures of interactions. These structures are organised in a category ***SIGN*** (of signatures) whose morphisms capture "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. ***SIGN*** can be proved to be finitely co-complete, which allows us to use colimits to express composition.

The other structure that is important for interaction protocols is that of the glues; because we are working with an institution [12], glues can themselves be organised in a category ***IGLU*** and a functor ***sign:IGLU*** $\rightarrow$ ***SIGN*** returns, for every glue, the structure of interactions (signature) that are being coordinated by the protocol. As a consequence, a morphism $\sigma: G_1 \rightarrow G_2$ of glues captures 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 ***sign***(G_1) as a part of ***sign***(G_2). ***IGLU*** is also a finitely co-complete category, meaning that we can use colimits to compose interaction protocols. Basically, colimits compute unions of specifications. We also know that $\boldsymbol{sign}_{IGLU}$ is a functor that makes ***IGLU*** coordinated over ***SIGN*** in the sense of [3]. We denote by ***iglu*** its left-adjoint, which returns an "empty" glue, i.e. one that does not introduce any requirements on the way interactions need to be coordinated.

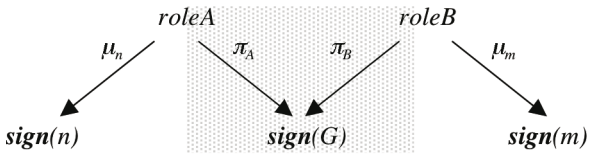
In this formal setting, every interaction protocol P consists of an interaction glue G and two signature morphisms $\pi_A: roleA \rightarrow \boldsymbol{sign}_{IGLU}(G)$ and $\pi_B: roleB \rightarrow \boldsymbol{sign}_{IGLU}(G)$. That is, an interaction protocol is a structured co-span in the sense of [8]:



Because a wire interconnects two parties of the module, we need some means of relating the interaction protocols used by the wire with the specifications (business roles or protocols) of the parties. The connection for a given party 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 ***sign***(n) associated with the node. These morphisms correspond to

the mappings defined by the rows of the tables that define the connector, as discussed in Section 3.2.

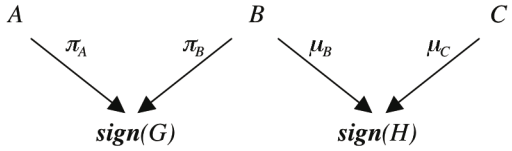
In this formal setting, a *connector* for a wire $n \leftrightarrow m$ between entities n and m in a module, is a structure $<\mu_n, \pi_A, G, \pi_B, \mu_m>$ where $<\pi_A, G, \pi_B>$ is an interaction protocol P and $<\mu_n, \mu_m>$ are the morphisms that connect the roles of P to the entities n and m. Such a connector defines the following diagram in ***SIGN***:



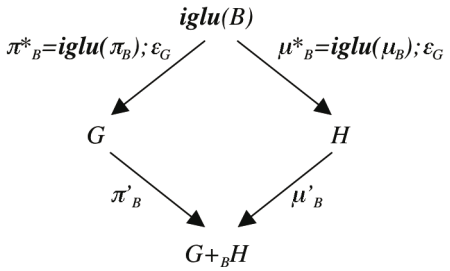
The interaction protocol $<\pi_A, G, \pi_B>$ corresponds to the shadowed part of the diagram. Given this, we take a module M to consist of:

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

An advantage of this algebraic characterisation is that we can easily explain how interaction protocols can be composed in support for run-time service discovery and binding. If we consider two interaction protocols with a common role:



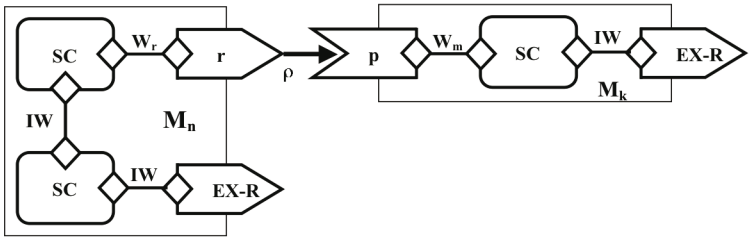
we compute the following pushout in ***IGLU***:



We define the composition of $<\pi_A,G,\pi_B>$ and $<\mu_B,H,\mu_C>$ to be $<\pi_A;\textbf{sign}(\pi'_B),G+_BH,\mu_C;\textbf{sign}(\mu'_B)>$.

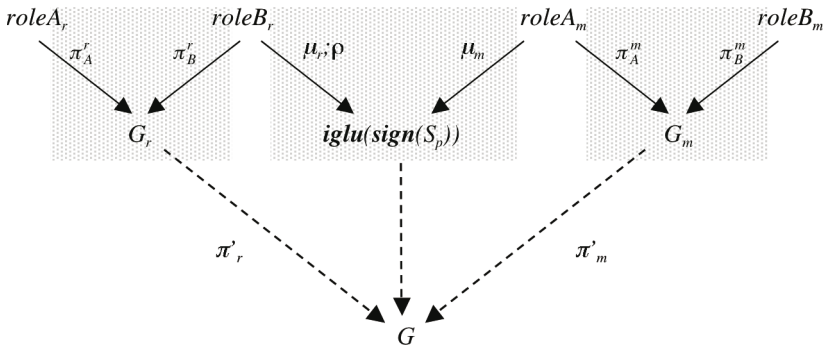
Consider now module composition. A binding between modules M_n and M_k consists of:

- A node $r \in requires(M_n)$, i.e. one of the requires-interfaces of M_n. Let this node be labelled with a business protocol S_r.
- A morphism $\rho:\textbf{sign}(S_r) \rightarrow \textbf{sign}(S_p)$ where S_p is the business protocol of $provides(M_k)$, i.e. of the provides-interface of M_k, such that all the properties required by S_r are entailed by those provided by S_r.

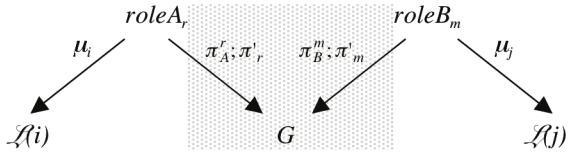


The module M that results from this process is defined by composing the wires W_r and W_k through the morphism ρ. This is achieved through the composition of the three co-spans that correspond to the interaction protocols of the wires W_r and W_k and, between them, the "external wire" established by the morphism ρ. Formally, the glue of this external wire, which is returned by the free functor ***iglu***, is "empty" in the sense that the protocol reduces to the syntactic binding established by the morphism.

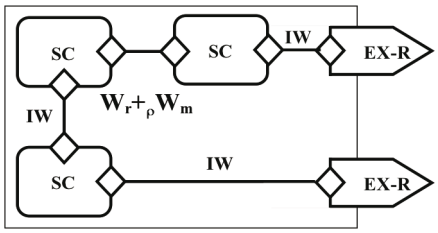
This composition is defined by the following diagram:



A new connector is defined by the composition of the morphisms that connect the roles to the new interaction glue:



This connector is now used for the wire that results from the composition:



4 Concluding Remarks and Further Work

In this paper, we presented the approach that we are developing within the SENSORIA project for modelling complex services. More precisely, we focused on the way we specify the protocols that are used for coordinating the interactions among the different parties that compose a service. This includes a logic adapted from $\mu UCTL$, a formalism being developed within SENSORIA for supporting qualitative analysis [11]. Our version of the logic uses a richer language of events that results from a conversation model of interactions: interactions are not specified in terms of pre and post-conditions but, rather, on properties that concern transactional behaviour, including pledges, deadlines and compensations. We are currently working on the axiomatisation of the primitives that capture such properties based on a semantic domain of doubly-labelled transition systems. We are also investigating the use of the 'on the fly' model checker UMC for supporting verification and validation [10].

Another important aspect of our model is an algebraic semantics that accounts for interaction protocols as structured co-spans, the full mathematical characterisation of which can be found in [8]. In the paper, we illustrated how this semantics provides a model for the composition of interaction protocols, connectors and wires, which is required for service discovery and binding.

In this paper, we addressed almost only the functional properties of service behaviour. The exception was the *delay* parameter that is associated with every wire. In fact, the composition of wires involves non-functional properties: for instance, we have $(W_r +_\rho W_m).delay = W_r.delay + W_m.delay$ because the external wire corresponding to ρ has no delay – it just binds names. Other non-functional properties are addressed in another report [5], including a constraint-based approach to SLAs.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services. Springer, New York (2004)
2. Baïna, K., Benatallah, B., Casati, F., Toumani, F.: Model-driven web service development. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 290–306. Springer, Heidelberg (2004)
3. Fiadeiro, J.L.: Categories for Software Engineering. Springer, New York (2004)
4. Fiadeiro, J.L.: Designing for software's social complexity. IEEE Computer 40(1), 34–39 (2007)
5. Fiadeiro, J.L., Lopes, A., Bocchi, L.: The SENSORIA Reference Modelling Language: Primitives for Configuration Management (2006) Available from www.sensoria-ist.eu
6. 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, Heidelberg (2006)
7. Fiadeiro, J.L., Lopes, A., Bocchi, L.: Algebraic semantics of service component modules. In: Fiadeiro, J.L., Schobbens, P.Y. (eds.) Algebraic Development Techniques, pp. 37–55. Springer, Heidelberg (2007)
8. Fiadeiro, J.L., Schmitt, V.: Structured co-spans: an algebra of interaction protocols. In: CALCO'07. LNCS. Springer, Berlin, Heidelberg, New York (In print 2007)
9. Foster, I., Kesselman, C. (eds.): The Grid 2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco, CA (2004)
10. Gnesi, S., Mazzanti, F.: On the fly model checking of communicating UML state machines. In: Second ACIS International Conference on Software Engineering Research, Management and Applications (SERA2004), pp. 331–338 (2004)
11. Gnesi, S., Mazzanti, F.: A model checking verification environment for UML Statecharts. In: Proceedings of XLIII Congresso Annuale AICA Comunita' Virtuale dalla Ricerca all'Impresa dalla Formazione al Cittadino. University of Udine – AICA (2005) (paper available from fmt.isti.cnr.it)
12. Goguen, J., Burstall, R.: Institutions: abstract model theory for specification and programming. Journal ACM 39(1), 95–146 (1992)
13. SCA Consortium (2005) Building Systems using a Service Oriented Architecture. Whitepaper available from www-128.ibm.com/developerworks/library/specification/ws-sca/