A Model-Checking Approach for Service Component Architectures*

João Abreu[1], Franco Mazzanti[2], José Luiz Fiadeiro[1], and Stefania Gnesi[2]

[1]Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
{jpad2,jose}@mcs.le.ac.uk
[2]Istituto di Scienza e Tecnologie dell'Informazione A. Faedo, CNR
Via G. Moruzzi 1, 56124, Pisa, Italy
{franco.mazzanti,stefania.gnesi}@isti.cnr.it

Abstract. We present a strategy for model-checking the correctness of service composition. We do so in the context of SRML, a formal modelling framework for service-oriented computing being defined within the SENSORIA project. We introduce a methodology for encoding patterns of typical service interaction with UML state machines and present a strategy for checking SRML specifications of service composition based on such patterns. For that purpose, we use the action-state branching time temporal logic UCTL and the model-checker UMC.

1 Specifying service composition with SRML

The SENSORIA Reference Modelling Language (SRML) [**?**,**?**] is a domain specific language for service-oriented architectures, inspired by the Service Component Architecture [**?**]. SRML provides primitives for modelling composite services whose business logic involves the orchestration of interactions among elementary components and the invocation of services provided by external parties.

Fig.1 is an example of a *service module* – the primitive that SRML offers for modelling service composition. A service module defines a distributed orchestration of a set of external services through a configuration of components and wires. Each of these components, wires and external services is typed by a specification of the interactions it can engage in or coordinate (in the case of wires). Components are typed by stateful models of the behaviour of the actual components that will execute during service delivery. Requires-interfaces, which represent the interfaces of the external services, are typed by what we call *business protocols* — behavioural constraints defined with patterns of the UCTL temporal logic [**?**] that need to be matched by the behaviour offered by the external services. Every service module has a *provides-interface* that is also typed by a business protocol advertizing the properties offered by the service at its interface level — in the example, the provides-interface CR is typed by the

* This work was partially sponsored through the IST-2005-16004 Integrated Project SENSORIA: Software Engineering for Service-Oriented Overlay Computers

business protocol *Customer* shown in Fig. 2. Finally, wires are typed by connectors that coordinate the interactions between components and external services [**?**]. A service module is said to be correct if the composition of components, wires and external-services that it specifies entails the properties advertized by its provides-interface.

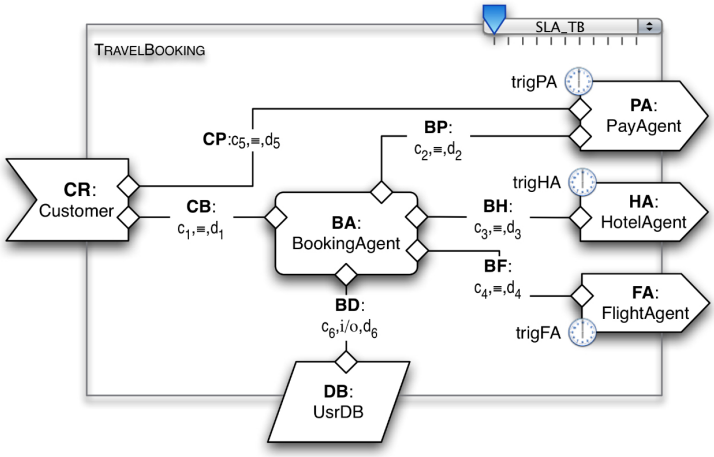

Fig. 1. The service module *TravelBooking*. *TravelBooking* uses the components *BA* and *DB* plus a set of wires to orchestrate three existing independent services — for booking a flight, booking a hotel and processing the payment.

Interactions, which have a conversational nature, consist of an asynchronous exchange of typed events between the parties that compose the service, where each type of event has a particular meaning from the business point of view (like requesting, replying, commiting, revoking, etc.). Service modules are interpreted over a particular type of Doubly Labelled Transition Systems (L^2TS) in which transitions are labelled by the publication, execution and discard of events [**?**] — UCTL logic is used to reason about such L^2TSs.

2 Specifying service interfaces with SRML

In SRML, the properties that are required from the external services that form the module, and also the properties that the module provides, are expressed through a business protocol in two ways: by declaring a set of typed interactions and by declaring a set of constraints that correlate the events of those interactions. The type that is associated with each interaction defines not only the set of events the service can engage in as part of that interaction, but also the conversational protocol that the service follows to engage in those events. The additional constraints that are specified in the business protocol – the behaviour

```
BUSINESS PROTOCOL Customer is
  INTERACTIONS
    r&s login
      🔔 usr:username, pwd:password
    r&s bookTrip
      🔔 from,to:airport,
         out,in:date
      ✉ fconf:fcode,
         hconf:hcode,
         amount:moneyvalue
    snd payNotify
      🔔 status:boolean
    snd refund
      🔔 amount:moneyvalue
  BEHAVIOUR
    initiallyEnabled login🔔?
    login✉! ∧ login✉.Reply enables bookTrip🔔?
    bookTrip✓? ensures payNotify🔔!
    payNotify🔔! ∧ payNotify.status enables bookTrip✟?
    bookTrip✟? ensures refund🔔!
```

Fig. 2. The business protocol *Customer*, which types the provides-interface *CR*.

– are used to impose further restrictions on that conversation or to correlate different interactions.

In order to specify behaviour constraints, SRML relies on a set of pre-defined patterns of behaviour that are encoded by abbreviations of UCTL formulas. The following table presents the abbreviations that encode three of the most commonly used patterns, which have been identified in a number of case studies:

initiallyEnabled e	$A\left(true_{\{\neg e¿\}} W_{\{e?\}} true\right)$
a enables e	$\left(AG[a]\neg EF < e¿ > true\right) \wedge \left(A[true_{\{\neg e?\}} W_{\{a\}} true\right)$
a ensures e	$\left(AG[a]AF[e!]true\right) \wedge \left(A[true_{\{\neg e!\}} W_{\{a\}} true]\right)$

The abbreviation "*initiallyEnabled* e" states that the event e will never be discarded (until it is actually executed) — this abbreviation is typically used to define the first interaction to take place during a session with a service. For instance *Customer* (shown in Fig. 2), which specifies the provides-interface of *TravelBooking*, declares that the request-event *login*🔔 is ready to be executed as soon as a session is created. The abbreviation "*a enables e*" states that after a happens the event e will not be discarded and that before a it will never be executed. In *Customer* this pattern is used to declared that, after the login is accepted (but not before), the service will be ready to execute a request to book a trip. Finally the abbreviation "*a ensures e*" states that after a happens the event e will for certain be published, but not before. This abbreviation is used in *Customer* to declare that after a request to revoke a booking is executed (but not before), a refund will be sent.

In the interaction declaration of a business protocol, two-way interactions can be typed by $s\&r$ (send and receive) or $r\&s$ (receive and send) to define that

the service being specified engages in the interaction as the requester or as the supplier, respectively. Each of these two roles, requester and supplier, has a set of properties associated with it. The following table presents the UCTL encoding of two of the properties associated with an interaction i of type $r\&s$.

A reply will be published after and only after the request-event was executed.	i♠? ensures i✉!
A revoke cannot be enabled before the execution of the commit-event.	$A[true_{\{\neg i\text{✞}_{?}\}}W_{\{i\checkmark_{?}\}}true]$

3 Encoding service composition with state machines

In order to be able to model-check properties of service behaviour in the context of SRML in general, and the correctness of service modules in particular, we need to restrict ourselves to those modules in which state machines are used for modelling the components, the wires and the behaviour required from external services. This is because the UMC model-checker [**?**] takes as input a system of UML communicating state machines, with which it associates a L^2TS that represents the possible computations of that system — model-checking is then performed over this L^2TS using UCTL logic. Using UML state machines for defining workflows is quite standard. However, the case of wires and requires-interfaces is not as simple. In the case of wires, we need to ensure that the SRML computational model [**?**] is adhered to in what concerns event propagation and related phenomena and in the case of requires-interfaces, we need to be able to represent the patterns discussed in the previous section with state machines.

Encoding requires interfaces A business protocol, which specifies the interface behaviour of a service, defines not one particular service, but a family of services that can be discovered, ranked and selected [**?**]. By associating a specific state machine with a requires-interface we are choosing a canonical model of the required behaviour.

As discussed in the previous section the specification of a requires-interface consists of a typed declaration of the interactions that the selected service should be ready to engage in and a set of behaviour constraints that correlate the events of those interactions. Our strategy for encoding a requires-interface as a state machine entails creating a concurrent region for each of the interactions that the external service is required to be involved in – the interaction-regions – and a concurrent region for all of the behaviour constraints – the constraint-regions – except for the constraints defined with the pattern "*initiallyEnabled e*": these are modelled by the instantiation of a state attribute.

The role of each of the interaction regions is to guarantee that the conversational protocol that is associated with the type of the interaction is respected. Events of a given interaction are published, executed and discarded exclusively by the interaction-region that models it. The role of the constraint-regions is to flag, through the use of special state attributes, when events become enabled

and when events should be published – the evolution of the interaction-regions, and thus the actual execution, discard and publication of events, is guarded by the value of those flags. Constraint-regions cooperate with interaction-regions to guarantee the correlation of events expressed by the behaviour constraints.

Following this methodology, each interaction declaration and each behaviour constraint encodes part of the final state machine in a compositional way. Associated with each interaction type and each constraint pattern, there is a particular statechart structure that encodes it. A complete mapping from interactions types and behaviour patterns to their associated statechart structure can be found in [**?**]. Naturally, the encoding we propose for specifications of requires-interfaces is defined so that the transition system that is generated for a service module satisfies the UCTL formulas associated the requires-interfaces of that module.

Encoding wires In SRML, the coordination of interactions, which are declared locally for each party of the module, is done by the wires. For each wire, there is a connector that defines an interaction protocol with two roles and binds the interactions declared in the roles with those of the parties at the two ends of the wire [**?**]. With our methodology for encoding wires with UML state machines, every connector defines a state machine for each interaction. This state machine is responsible for transmitting the events of that interaction from the sending party to the receiving co-party. Parties publish events by signalling them in the state machine that corresponds to the appropriate connector; this state machine in turn guarantees that these events are delivered by signalling them in the state machine that is associated with the co-party. The relation between parameter values that is specified by the interaction protocol of the connector is ensured operationally by the state machine that encodes that connector – data can be transformed before being forwarded. The statechart contains a single state and as many loops as the number of events that the connector has to forward.

4 Model-Checking service modules: the *TravelBooking* example

In order to model-check that the composition specified by the module *TravelBooking* provides the properties specified in *Customer*, we have encoded each of its external-required interfaces and each of its connectors using the methodology described in the previous section. Adding the two components that orchestrate the system, we ended up with a set of fifteen communicating UML state machines. Because every input source of a UMC model must also be modelled via an active object, we had to define a machine that initiates the interactions advertised in the provides-interface *Customer*, thus modelling a generic client of the service. Using this system as input to the UMC model-checker, we can verify if the doubly labelled transition system that is generated — we will refer to it as T — does satisfy the formulas associated with the provides-interface *Customer*, shown in Fig. 2. If T does not satisfy some of these formulas, than there is something in the module *TravelBooking* that needs to be corrected.

Having used UMC to model-check *TravelBooking*, we found out that all the constraints were satisfied by T except one: "*payNotify*♠*!* $\wedge$ *payNotify.status enables bookTrip*♆*?*". This is because there is a path in T on which the event *bookTrip*♆ is discarded after the event *payNotify*♠ is published with a positive value for the *payNotify.status* parameter. This means that the publication of event *payNotify*♠ with a positive *payNotify.status* by the service does not guarantee that the revoke event of interaction *payNotify* becomes enabled for execution. If the composition was implemented as it is, it would be possible for a client to ask for a booking to be revoked and have this request ignored by the service.

After analysing the path of T that leads to the failure of the property, we understood that the problem is that, because *PA* interacts directly with the client through the wire *CP*, it is possible for the payment notification (represented by *payNotify*♠) to be received by the client before *BA* receives the confirmation for the payment (which is sent via another wire, *BP*). If the client tries to revoke the booking immediately, *BA* will not accept it because it does not yet know that the payment of the booking has been accepted by *PA*.

In order to fix this problem we have redesigned the architecture of the module *TravelBooking* by removing the wire *CP*. In the new architecture, *PA* does not interact directly with the client anymore. When the payment is executed by *PA*, the component *BA* is notified and is in turn responsible for notifying the client. Only then can the client choose to revoke the booking.

Acknowledgements

We would like to thank Antónia Lopes and Laura Bocchi for helping us stay on the right path (and states).