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 behavioural correctness of complex services within service-oriented architectures. 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-oriented behaviour with UML state machines and present a strategy for checking SRML specifications of complex services based on such patterns. For that purpose, we use the action-state branching time temporal logic UCTL and the UML state machine based model-checker UMC.

1 Introduction

SENSORIA [18] is an integrated project funded under the FET Global Computing initiative, which is developing a software engineering approach to service-oriented computing (SOC) based on formal foundations. In this context, a number of languages and calculi are being investigated that address different levels of abstraction of the software engineering process. In this paper, we are concerned with the SENSORIA Reference Modelling Language – SRML [13] – which provides primitives for modelling composite services whose business logic involves the orchestration of interactions among more elementary components and the invocation of services provided by external parties. SRML is inspired by the Service Component Architecture – SCA [21] – and makes available a modelling framework that is independent of the languages and platforms that are currently being provided for web [4] (or grid [15]) services.

In Fig.1, we provide an example of a *service module* – the primitive that SRML offers for modelling complex services. In a nutshell, a service module prescribes a distributed orchestration for a type of services through a configuration of components and wires. The components can be internal to the service (in the sense that they are created each time the service is invoked and killed

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

when the service terminates), in which case they are represented in the module by what we call a *component-interface* (like BA). They can be persistent components (like databases) that are part of the business environment in which the service operates, in which case they are represented in the module by what we call a *uses-interface* (like DB). Modules may also declare a number of (external) services that need to be discovered and bound on the fly to the other components, in which case they are represented in the module by what we call a *requires-interface* (like PA, HA and FA).

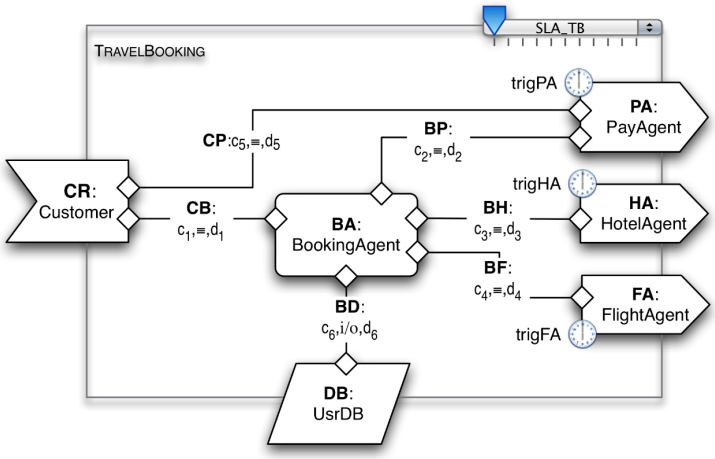

Fig. 1. The service module $TravelBooking$.

Component- and uses-interfaces are typed by models of the behaviour of the actual components that will execute during service delivery. Requires-interfaces are typed by what we call business protocols – patterns of behaviour that need to be matched by the behaviour offered by the discovered services. Every service module has a *provides-interface* (like CR) that is also typed by a business protocol declaring the properties that can be expected from the behaviour of the service. Hence, every instance (session) of service *TravelBooking* is assembled by connecting an internal component of type *BookingAgent* to a persistent component (a database of users) of type *UsrDB* and, when required, to external services of type *PayAgent*, *HotelAgent* and *FlightAgent*.

Service modules also declare service-level agreement (SLA) constraints (indicated in the example as SLA_TB) that are used for service selection and ranking, as well the conditions that trigger the discovery of external services (indicated in the example as $trigPA$, $trigHA$ and $trigHA$). Such dynamic and non-functional aspects of SOC are outside the scope of this paper (see [14] instead).

Our purpose in this paper is to present an approach for analysing the properties that can emerge from the orchestration of service behaviour in general, and

the correctness of service modules in particular, i.e. the property that establishes that the properties declared in the provides-interface of a module are actually guaranteed by the orchestration performed by the configuration of components and wires on the assumption that the external services that are discovered and bound to components satisfy the requires-interfaces. Several approaches have been proposed for the verification of service-oriented systems, essentially for orchestration languages like BPEL [25] or in the context of service publication and discovery (e.g. [23]). However, such approaches are usually based on concepts like pre- and post-conditions, which offer specifications of service behaviour that are confined to static/transformational aspects of black-box behaviour that only take into account initial and final states of service execution. SRML adopts instead a sophisticated notion of two-party interaction that is conversational in the sense that it involves a number of correlated events that need to obey a well-defined protocol. These events include typical notions of service-oriented business protocols such as committing to, cancelling, and revoking deals.

This is why we decided to adopt instead verification techniques used for concurrent and distributed processes, namely model-checking of temporal specifications that can capture the conversational properties of services. The particular approach that we adopted is based on the model-checker UMC [19], which is being used in SENSORIA in conjunction with service calculi such as COWS [12]. UMC works over UML state machines and UCTL [5], a temporal logic that is interpreted over transition systems in which both states and transitions are labelled, thus making it easier to express properties of stateful interactions as required by SRML. One of the novelties of our work, and one of the main contributions that we make to SOC in general, is in the way we use these model-checking techniques for analysing service modules. The particular challenges that we address in the paper concern:

- The encoding of the conversational protocols that service interactions are required to exhibit as UML state machines and UCTL properties;
- The encoding of the behavioural patterns used in business protocols again as UML state machines and UCTL properties.

The double encoding, as UML state machines and UCTL, is justified by the fact that UML state machines are used for expressing the (distributed) orchestration defined by the module and UCTL is used for verifying properties of the service thus orchestrated. More precisely, in what concerns the former, the strategy is to use UML state machines as models for the internal components, the persistent components, the external services and the connecting wires.

In summary, the main emphasis of the paper is on the way our model-checking approach addresses the notion of interaction available in SRML, the methodology of encoding patterns of typical service-oriented behaviour with UML state machines, and the strategy for checking the correctness of SRML service modules based on such patterns. In section 2 we give an overview of the background material that underlies the approach: SRML as a modelling approach, the logic UCTL and the model-checker UMC. In section 3 we explain how we use the logic

UCTL to reason about service-oriented architectures within the SRML framework. Section 4 presents our methodology for encoding a SRML module as a set of communicating UML state machines. In section 5 we illustrate how our approach works in practice by model-checking the module $TravelBooking$ (see Fig.1). Finally, section 6 wraps up the paper and discusses further work.

2 Background

2.1 Modelling interactions in SRML

As already mentioned, SRML models complex services in terms of two-way interactions that capture a pattern of dialogue that is prevalent in service-oriented systems: a party sends a request to a co-party that replies either positively by making a pledge to deliver a set of properties, or negatively, in which case the interaction ends; if the answer is positive the party that made the request can commit by accepting the pledge or refuse the pledge and cancel the interaction. If and after the requester commits, a revoke may be available that compensates for the effects of the pledge. The set of events associated with an interaction a is shown in the following table:

a♠	The initiation-event of a.
a✉	The reply-event of a.
a✓	The commit-event of a.
a✗	The cancel-event of a.
a♆	The revoke-event of a.

Interactions are peer-to-peer between pairs of entities connected through wires – *CB*, *CP*, *BP*, *BH*, *BF*, and *BD* are the wires in *TravelBooking*. Wires are typed by connectors, which provide a model of the protocol that coordinates the interactions between the two parties. This is indicated in the module through triples of the form $\langle c, P, d\rangle$ where P is an interaction protocol and c and d attach the protocol to the parties at the ends of the wires. By $\equiv$ we denote a straight interaction protocol [2] that binds together two interactions. For example, the existence of a a straight protocol between interactions $bookHotel$ (declared in $BookingAgent$) and $lockHotel$ (declared in $HotelAgent$) means that the wire simply forwards the events of these interactions.

Parties engage in interactions independently of their co-parties, i.e. the workflow that determines when a party interacts, by publishing an event or processing it, is independent of the way these events are transmitted. Transmission of events follows a number of phases, independently of the protocol enforced by the wire: when an event is published by a party, it is picked by the wire for delivery, which is subject to a delay (negotiable as part of an SLA). Once delivered (according to the interaction protocol of the connector), it is eventually processed by the co-party, which either executes or discards it, depending on whether the event is enabled or not in the current state of the co-party. We use $e!$ to refer to the action of publishing event e, $e?$ to refer to the action of executing it and $e¿$ to refer to

the action of discarding it. These actions are used for expressing and reasoning about properties of services as discussed further on. More detailed information on the computational model of SRML can be found in [3].

Complex services like *TravelBooking* establish multi-party collaborations by orchestrating their interactions. Orchestration is performed in a distributed way by the given configuration of components and wires. Each component provides a model (called business role) of the local orchestration that it performs over the events that the component can engage in. SRML provides a declarative orchestration language based on states and transitions that allows for underspecification of the effects that transitions have on states and of the events that are published [13]. This is because SRML is being defined to support a design methodology in which service modules can accommodate different levels of development.

SRML also supports a process of stepwise refinement that can lead to business roles that are executable, in which case it supports the use of notations and languages that are more procedural such as BPEL (for which a translation into SRML has been defined [6]) and, in the case of this paper, UML state machines. The same applies to the interaction protocols that model the coordination performed by the connectors. In Fig.2 we show the UML state machine (business role) corresponding to the workflow associated with *BookingAgent*.

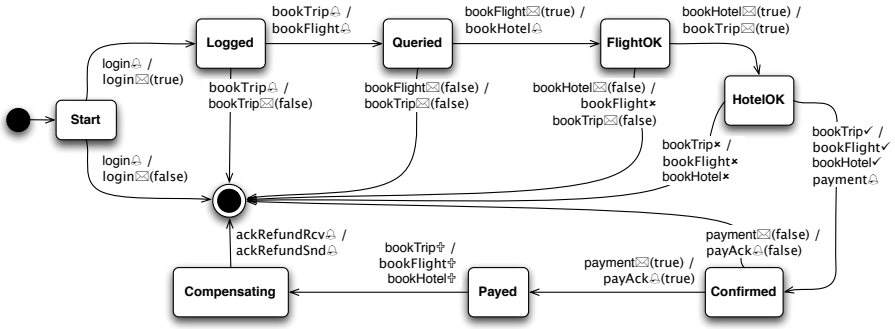

Fig. 2. The statechart that models the orchestration performed by *BookingAgent*. Parameters that do not affect the workflow are not shown.

We have already mentioned that requires- and provides-interfaces specify properties that involve a number of patterns, which in SRML are expressed in the language of business protocols. In Fig.3 we present the business protocol that specifies the requires-interface of FA. In sections 3 and 4, we formalise these patterns in terms of the logic UCTL and UML state machines, respectively. The idea behind this dual view is that, for provides-interfaces, we use the UCTL encoding and, for requires-interfaces, we use the UML state machine encoding so that, in conjunction with the UML state machines that correspond to the business roles (components) and interaction protocols (wires) we have a set of

communicating UML state machines over which we can model-check the UCTL encoding of the provides-interface.

```
BUSINESS PROTOCOL FlightAgent is
  INTERACTIONS
    r&s lockFlight
      🔔 from,to:airport,
         out,in:date,
         traveller:usrdata
      ✉ fconf:fcode
         amount:moneyvalue,
         beneficiary:accountn,
         payService:serviceId
    rcv payAck
      🔔 proof:pcode
         status:bool
    snd payRefund
      🔔 amount:moneyvalue
  BEHAVIOUR
    initiallyEnabled lockFlight🔔?
    lockFlight✉! ∧ lockFlight✉.Reply enables payAck🔔?
    payAck🔔? ∧ payAck🔔.status enables lockFlight✟?
    lockFlight✟? ensures payRefund🔔!
```

Fig. 3. The specification of the requires-interface *FlightAgent* written in the language of business protocols. Services of this type are required to be involved in three kinds of interaction with a number of parameters that carry data required for booking a flight and processing payments. Some properties of this interaction are specified: a flight booking can be initiated once a session of the service is created, the service will accept an acknowledgment of payment after (and only after) sending a positive reply to the customer, a flight reservation can be revoked anytime after payment has been confirmed, and revoking a booking guarantees a refund.

2.2 UCTL and UMC

Recently various logics have been introduced ([7, 17] are examples) that allow one to reason about both action-based and state-based properties. The advantage of all these logics lies in the availability of operators that make it easier to formulate properties that, in pure action-based or pure state-based logics, can be quite cumbersome to write down. This is especially useful when logics are to be used in conjunction with languages and notations that–like the UML–allow both action and state changes to be expressed. In such cases, the use of combined action and state operators has the additional advantage of often leading to a reduced state space, smaller memory, and less time spent during verification.

In this paper, we will use UCTL [5], which includes both the branching-time action-based logic ACTL [9] and the branching-time state-based logic CTL [11]. The models of UCTL are doubly labelled transition systems (L^2TS for short) which are transition systems whose states are labelled by atomic propositions and whose transitions are labelled by sets of actions [10]. The syntax of UCTL

formulas is defined as follows:

$$\phi ::= true \mid p \mid \phi \wedge \phi' \mid \neg\phi \mid E\pi \mid A\pi$$
$$\pi ::= X_\chi \phi \mid \phi \,{}_\chi U \,\phi' \mid \phi \,{}_\chi U_{\chi'} \,\phi' \mid \phi \,{}_\chi W \,\phi' \mid \phi \,{}_\chi W_{\chi'} \,\phi'$$

where p ranges over *state predicates*, χ over *actions*, ϕ over *state formulae*, and π over *path formulae*. E and A are "exists" and "for all" *path quantifiers* respectively. The next operator X says that in the next state of the path, reached by an action satisfying χ, the formula ϕ holds. The intuitive meaning of the doubly-indexed until operator U on a path is that ϕ' holds at some future state of the path reached by a last action satisfying χ', while ϕ has to hold from the current state until that state is reached and all the actions executed in the meanwhile along the path either satisfy χ or τ. Finally, the weak until operator W holds on a path either if the corresponding strong until operator holds or if for all states of the path the formula ϕ holds and all the actions of the path either satisfy χ or τ. It is straightforward to derive the well-known temporal logical operators EF ("possibly"), AF ("eventually") and AG ("always") and the diamond and box modalities <> ("possibly") and [] ("necessarily"). In particular, $< \chi > \phi$ stands for $EX_\chi \phi$, meaning that there is transition that satisfies χ which leads to a state that satisfies ϕ; and $[\chi]\phi$ stands for $\neg < \chi > \neg \phi$, meaning that every transition that satisfies χ leads to a state that satisfies ϕ.

UMC [19] is an on-the-fly model-checker developed for efficient verification of UCTL formulae over a set of communicating UML state machines [22]. A UMC model description consists of a set of UML class definitions and a static set of object instantiations – the actual state machines that form the system under analysis. A UMC model must represent an input-closed system, i.e. the input sources must be modelled as active objects interacting with the rest of the system. Each state machine has a pool that buffers the set of signals that have been received from other machines and are waiting to be processed by that machine. According to its class definition, each state machine has at any given time a set of values for each local attributes and a set of currently active sub states as specified by the statechart diagram of the class.

3 The UCTL patterns of service-oriented behaviour

SRML models correspond to L^2TSs in which the several stages of events propagation (publish, deliver, execute or discard) are the actions that label the transitions, and the pledges (i.e. the properties that hold after positive replies) and the history of events are the state predicates (the history of events is modelled as state predicates because UCTL does not have past operators).

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 external 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 – are used to impose further restrictions on that conversation or to correlate events of different interactions. Both types of properties need to be model-checked for the provides-interface (*Customer* in the running example) because they depend on the correct orchestration of the components when connected to the required services. Therefore, both need to be encoded as UCTL formulas. We will first address the encoding of the patterns that are used to specify behaviour constraints and then we will address the encoding of the conversational protocol that is associated with the interaction types.

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

Fig. 4. The provides-interface *Customer*.

3.1 Behaviour constraints

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. These patterns capture how the events of different interactions are typically correlated in service-oriented architectures. The following table presents the abbreviations that encode three of the most commonly used patterns that we have 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 figure 4), the provides-interface of *TravelBooking*, declares that the 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.

3.2 The two-party interaction pattern

In the interaction declaration of a business protocol, two-way interactions are 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 two tables, present the UCTL encoding of some of the properties associated with the types $s\&r$ and $r\&s$, respectively.

$s\&r$ — Requester	
The reply-event becomes enabled by the publication of the initiation-event and not before.	i♠! enables i✉?
$r\&s$ — Supplier	
The reply will be published after and only after the initiation-event was executed.	i♠? ensures i✉!
The revoke-event cannot be enabled before the execution of the commit-event.	$A[true_{\{\neg i\text{⇧}?\}}W_{\{i\checkmark?\}}true]$

The service module *TravelBooking*, declares, through its provides-interface *Customer*, that it engages in interactions *login* and *bookTrip* following the $r\&s$ protocol (notice that the type declaration in the provides-interface is actually done from the symmetrical point of view of the client). In order to verify the correctness of *TravelBooking*, the set of properties that is associated with the type $r\&s$ (two of which are shown in the table) will have to be model-checked for each of these two interactions.

4 From SRML Modules to UML state machines

We have already mentioned that, 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 restrict ourselves to those modules in which state machines are used for modelling the internal components, the persistent

components, the protocols performed by the wires, and the required behaviour of external services. This is because UMC takes as input a set of 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 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 as discussed in 2.1. In the case of requires-interfaces, we need to discuss how the patterns defined in section 3 can be represented with state machines.

4.1 Encoding external-required services

In SRML, requires-interfaces are specified, through business protocols, with the patterns of temporal logic that we discussed in section 3. Such a specification defines not one particular service, but a family of services that can be discovered, ranked and selected according to the way they optimise SLAs. By associating a specific state machine with a requires-interface we are choosing a canonical model of the required behaviour.

As illustrated in Fig. 3 and discussed in section 3, 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*": as discussed further ahead, 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 as discussed in section 3.2. 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 (discussed in section 3.1). We illustrate this methodology by presenting the encoding of the requires-interface *FlightAgent* in Fig. 5.

Following our methodology, each interaction declaration and each behaviour constraint encodes part of the final state machine in a compositional way. Associated with each interaction type, there is a particular statechart structure that encodes it. Each of the patterns of behaviour constraints is also associated with a particular statechart structure. A complete mapping from interactions types and behaviour patterns to their associated statechart structure can be found in

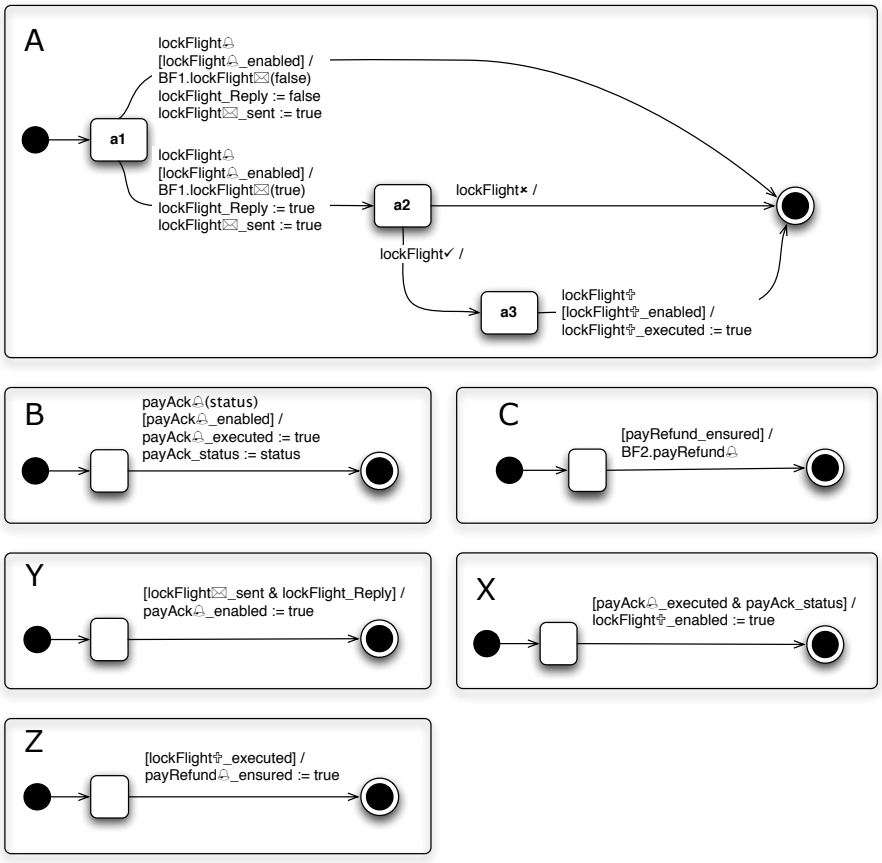

Fig. 5. The UML statechart encoding of the requires-interface *FlightAgent*. *FlightAgent* is involved in the three interactions *lockFlight*, *payAck* and *payRefund* that are encoded by interaction-regions A, B and C, respectively; these three interactions are correlated by four behaviour constraints that originate the three constraint-regions Y, X, and Z. The constraint "*initiallyEnabled lockFlight?*" does not originate a region in the state machine; instead it determines that the flag *lockFlight_enabled* is initially set to *true* and therefore when the event *lockFlight* is processed it will be executed (and not discarded) by interaction-region A. When *lockFlight* is executed, interaction-region A evolves from state $a1$ to state $a2$ by publishing a positive reply or alternatively from $a1$ to the final state by publishing a negative reply. When this reply is published the flag *lockFlight_sent* is set to *true* so that the other regions of the state machine are informed that this event was published — the constraint-region Y, for instance, that encodes the constraint "*lockFlight! ∧ lockFlight.Reply enables payAck?*" reacts to the publication of a positive reply by setting the flag *payAck_enabled* to true thus enabling the execution of event *payAck* by interaction-region B. When *payAck* is executed by interaction-region B the flag *payAck_executed* becomes true and as a consequence the constraint-region X evolves (if the parameter *payAck.status* is *true*) by setting *lockFlight_enabled* to true, thus enabling the execution of event *lockFlight* by interaction-region A. Finally, when *lockFlight* is executed the flag *lockFlight_executed* is set to *true* and as a consequence the interaction-region C will publish the event *payRefund*.

[1]. Naturally, the encoding we propose for specifications of requires-interfaces is defined in such a way that the transition system that is generated for a service module satisfies the UCTL formulas that are associated with each of the requires-interfaces of that module.

4.2 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 [2]. 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 signaling them in the state machine that corresponds to the appropriate connector; this state machine in turn guarantees that these events are delivered by signaling 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.

In the *TravelBooking* module two-way interactions are coordinated by straight interaction protocols that bind the names and parameters of $s\&r$ and $r\&s$ interaction declarations directly (i.e. events and parameter values are the same from the point of view of the two parties connected). Figure 6 shows the state machine that encodes this connector for the single interaction that takes place between *BA* and *HA* — there is only one persistent state in which the machine waits to receive events and forward them with the same parameter values.

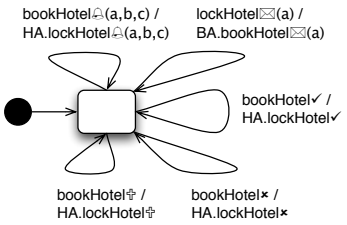

Fig. 6. The UML encoding of the connector that coordinates the single, two-way, interaction between *BA* and *HA* which is named *bookHotel* and *lockHotel* from the point of view of each party respectively.

5 Model-Checking the module *TravelBooking*

In order to model-check the module *TravelBooking* 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 properties associated with the provides-interface *Customer*, shown in figure 4. As discussed in section 3, these consists of the constraints associated with the types of the declared interactions (see section 3.2) and those that derive from the patterns of behaviour (see section 3.1). If T does not satisfy any of the previous 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*♠*!* ∧ *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 current system 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 *CR* through the wire *CP*, it is possible for the payment notification (represented by *payNotify*♠) to be received by *CR* before *BA* receives the confirmation for the payment (through *payment*✉). If *CR* 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 *CR* anymore. When the payment is executed by *PA*, the component *BA* is notified and is in turn responsible for notifying *CR*. Only then can the customer choose to revoke the booking.

6 Concluding Remarks and Further Work

In this paper, we have presented a strategy for model-checking properties of specifications of complex services in the SENSORIA Reference Modelling Language – SRML. We have focused in particular on the use of model-checking for validating the correctness of service modules, which corresponds to checking that the

properties declared in the provides-interface of the module (which corresponds to the description of the functional properties of the service) are enforced by the orchestration defined in the model provided that required services discovered and bound during the orchestration satisfy their requires-interfaces. For this purpose, we have used the UCTL temporal logic [5] for expressing behavioural properties of services and UML state machines to model their orchestration. For model-checking service modules with UMC [19] we have defined a methodology in which the behaviour of required-services is encoded with UML state machines. Our methodology poses no problems of scalability because the translation of modules into UML state machines is linear in their size and the complexity of the model checker is linear in the size of the model and in length of the formula being checked. The actual complexity of model-checking a SRML module clearly depends on the size of the generated state space, which results strictly from the interactions that can occur between the components of the module.

In the literature, we can find other formalisms for describing services, normally associated with an orchestration language like BPEL [25] or the use of repositories for service publication and discovery [23]. Dynamic logic [16] is often adopted, reflecting an approach to service description based on pre/post-conditions, i.e. on the transformation that they operate over data or state, much in the same way as for method specification in design by contract [20]. In our opinion, such approaches miss one of the distinguished aspects of service-oriented computing, which is the conversational nature of the interactions. This is why SRML provides a set of primitives for modelling such kind of interactions and we decided to adopt a branching-time temporal logic like UCTL for reasoning about their properties. One of the contributions of our work is to offer a more expressive means of service analysis that relies on a number of patterns of behaviour validated on several case studies. A more systematic study of useful patterns is underway that includes recent studies in workflow modelling [26]. The use of description logic as in [23] also serves an important role in service publication and discovery, as demonstrated by their use in support of the Semantic Web. Extensions of SRML are being planned in order to incorporate an ontological component that can support matchmaking.

The other advantage of SRML over approaches based on dynamic logic or process algebras [24] is that the models that we provide are more abstract: where dynamic logic and process algebras are compositional over program or process structures, SRML is compositional over the business structure. That is, the notion of service module in SRML reflects dependencies between services that derive from the way services need to fulfil business goals. As a consequence, our model-checking approach involves a number of UML state machines that reflect the roles that different entities play in service delivery.

Certain aspects of the approach that we have presented here are still being further investigated. From a theoretical point of view, we are developing a formal characterisation of the relationship between the encoding of the patterns in a UCTL and in UML state machines (beyond the fact that the latter model-checks the former, of course!). Another important investigation at the moment

concerns the relationship between the operational semantics that UMC implements for UML state machines and the computational model defined for SRML in [3]. From a pragmatic point of view, we are currently addressing the way parameters of SRML interactions can be handled by UMC. More precisely, we are considering adding to UMC the possibility of using symbolic parameters [8]. Another important aspect concerns the support for the primitives that SRML provides for modelling quantitative aspects of time in SOC, which the current version of UMC cannot handle. This should also lead us to consider stochastic approaches which are already being used in SENSORIA.

Acknowledgements

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

References

1. J. Abreu. A Formal Framework for Modelling Service Component Architectures. PhD thesis. Forthcoming.
2. J. Abreu, L. Bocchi, J. L. Fiadeiro, and A. Lopes. Specifying and Composing Interaction Protocols for Service-Oriented System Modelling. In *FORTE'07*, volume 4574 of *LNCS*, pages 358–373. Springer, 2007.
3. J. Abreu and J. L. Fiadeiro. A Coordination Model for Service-Oriented Interactions. In *Coordination'08*, volume 5052 of *LNCS*, pages 1–16. Springer, 2008.
4. G. Alonso, F. Casati, H. A. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, 2004.
5. M. Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. An action/state-based model-checking approach for the analysis of communication protocols for Service-Oriented Applications. In *FMICS'07*, LNCS. Springer-Verlag, Berlin, 2007.
6. L. Bocchi, Y. Hong, A. Lopes, and J. L. Fiadeiro. From bpel to srml: a formal transformational approach. In *Web Services and Formal Methods*, volume 4937 of *LNCS*, pages 92–107. Springer, 2008.
7. S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17(4):461–483, 2005.
8. E. M. Clarke, K. L. McMillan, S. V. A. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *CAV'96*, volume 1102 of *LNCS*, pages 419 – 427. Springer-Verlag London, UK, 1996.
9. R. De Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes*, pages 407–419, 1990.
10. R. De Nicola and F. W. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.
11. E.M. Clarke, E.A. Emerson and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
12. A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A model checking approach for verifying cows specifications. In *FASE'08*, volume 4961 of *LNCS*, pages 230–245. Springer, 2008.

13. J. L. Fiadeiro, A. Lopes, and L. Bocchi. A Formal Approach to Service Component Architecture. *Web Services and Formal Methods*, 4184:193–213, 2006.
14. J. L. Fiadeiro, A. Lopes, and L. Bocchi. An abstract model of service discovery and binding. Submitted. Available from www.cs.le.ac.uk/people/jfiadeiro, 2008.
15. I. Foster and C. K. Eds. *The Grid 2: Blueprint for a New Computing Infrastructure.* Morgan Kaufman, San Francisco, CA, 2004.
16. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* Foundations of Computing. MIT Press, 2002.
17. M. Huth, J. Agadeesan, and D. Schmidt. Modal transition systems : A foundation for three-valued program analysis. In *ESOP'01*, volume 2028 of *LNCS*, pages 155–169. Springer-Verlag, Berlin, 2001.
18. Martin Wirsing et al. SENSORIA: A systematic approach to developing service-oriented systems — white paper, 2007.
19. F. Mazzanti. UMC User Guide v3.3. Technical Report 2006-TR-33, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR. Available from http://fmt.isti.cnr.it/WEBPAPER/UMC-UG33.pdf, 2006.
20. B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 2000.
21. Michael Beisiegel et all. Service component architecture specifications, 2007.
22. Object Management Group. Unified Modeling Language. http://www.uml.org/.
23. C. Pahl. An ontology for software component matching. *Int. J. Softw. Tools Technol. Transf.*, 9(2):169–178, 2007.
24. G. Salaün, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *ICWS '04*, page 43. IEEE Computer Society, 2004.
25. F. van Breugel and M. Koshkina. Models and verification of bpel, 2006. Available from http://www.cse.yorku.ca/ franck/research/drafts/tutorial.pdf.
26. W. van der Aalst and M. Pesic. DecSerFlow: Towards a truly declarative service flow language. In *WS-FM*, pages 1–23, 2006.