Timed Automata for Web Services Verification *

M. Emilia Cambronero, Juan J. Pardo, Gregorio Díaz, Valentín Valero
Computing Systems Department
University of Castilla-la Mancha
Avda. España s/n 02071 Albacete
Spain

Abstract: Timed Automata is a well known formalism for the description of Real-Time System. In this paper we show that it is useful for modelling and analyzing Web Services with time constraints. For our purposes it is enough with the choreography level of the Web Services Architecture, so Web Services descriptions written in WSCDL (a XML-based description languages) are translated to timed automata models. Then, the UPPAAL tool is used to simulate and analyse the behavior of the system. The process is illustrated with a case study, an airline ticket reservation system. The example shows the systematic translation process, and the resulting model is checked for a suite of rather generic safeness and liveness properties.

Key–Words: WS-CDL, Timed Automata, Web Services,Verification, Model Checking

1 Introduction

Web Services are a key component of the emerging, loosely coupled, Web-based computing architecture. A Web Service is an autonomous, standards-based component whose public interfaces are defined and described using XML [6]. Other systems may interact with a Web Service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.

The Web Services specifications offer a communication bridge between the heterogeneous computational environments used to develop and host applications. The future of E-Business applications requires the ability to perform long-lived, peer-to-peer collaborations between the participating services, within or across the trusted domains of an organization.

Web Services cover a wide range of systems, which in many cases have strict time constraints, e.g., peer-to-peer collaborations may have time limits on completion. They are described in the top level layers of Web Services architectures [6] with elements such as *time-outs* and *alignments*. Time-outs allow each party to fix the time for an action to occur, while alignments are synchronizations between two peer-to-peer parties. Businesses rely on Web Services, it is therefore important that the Web Services framework ensure the correctness of systems with time constraints. Verification of real-time properties is especially important for Web sites that offer services with some kind of time restrictions, such a maximum times to keep a reservation of a flight seat, or maximum delays to order fund transfers, etc. Since model checking of timed automata has proven to be very useful for similar problems in other areas, we investigate, how it can be applied for Web Services.

This verification process starts from the top level layers of Web Services architectures [6] and more specifically from the choreography level, thus we use the Web Service Choreography Description Language (WS-CDL) [6] as input for this verification process. Therefore, the starting point are specification documents written in WS-CDL. Then, these descriptions are translated into timed automata, and the UPPAAL tool is used to simulate and verify the correctness of the system.

We illustrate the approach with a particular case study: an airline ticket reservation system, which contains some time constraints.

Related Work

There is a growing consensus that the use of formal methods, and the development of methodologies based on these formalisms, could have significant benefits in developing E-business systems, due to the enhanced rigor these methods bring. Some of these approaches have used logics to formalize and reason about the contracts specified by the description languages [1], [2] or [3]. They have the flexibility and expressive power of logics, but it is well known that

*This work has been supported by the CICYT project "Description and Evaluation of Distributed Systems and Application to Multimedia Systems",TIC2003-07848-C02-02.

in general such logics are not decidable, and thus verification requires expert knowledge. Closer to our approach are formalizations based on finite state machines [4, 8] and Petri Nets [5], they allow automated verification using model checking tools; but they differ from our work in not treating timing constraints. Our approach is based directly on the theory of timed automata [7], and we use the UPPAAL tool [10, 11, 12], although other similar tools like KRONOS [13] would apply.

Overview

The paper is structured as follows. In Section 2 we present the case study, and how it is described with WSCDL. Translation from WS-CDL to Timed automata is described in Section 3, and then in Section 4 we show how we can use this tool to model, simulate and verify the system behavior. Finally, conclusions and future work are presented in Section 5.

2 Case Study

Reservation System to verify the availability of seats (*Verify Seats Availability*). When *Order Trip* has been completed, the Traveller has the choice of accepting or rejecting the proposed itinerary, and he can also decide not to take the trip at all.

- In case he rejects the proposed itinerary, he may submit the modifications (*Change Itinerary*), and wait for a new proposal from the Travel Agent.
- In case he decides not to take the trip, he informs the Travel Agent (*Cancel Itinerary*) and the process ends.
- In case he decides to accept the proposed itinerary (*Reserve Tickets*), he will provide the Travel Agent with his Credit Card information in order to book the itinerary.

Once the Traveller has accepted the proposed itinerary, the Travel Agent connects with the Airline Reservation System in order to reserve the seats (*Reserve Seats*). However, it may occur that at that moment no seat is available for a particular leg of the trip, because some time has elapsed from the moment in which the availability check was made. In that case the Travel Agent is informed by the Airline Reservation System of that situation (*No seats*), and the Travel Agent informs the Traveller that the itinerary is not possible (*Notify of Cancellation*). When the reservation is made, the Travel Agent informs the Traveller (*Seats Reserved*). This reservation is valid fr one day,

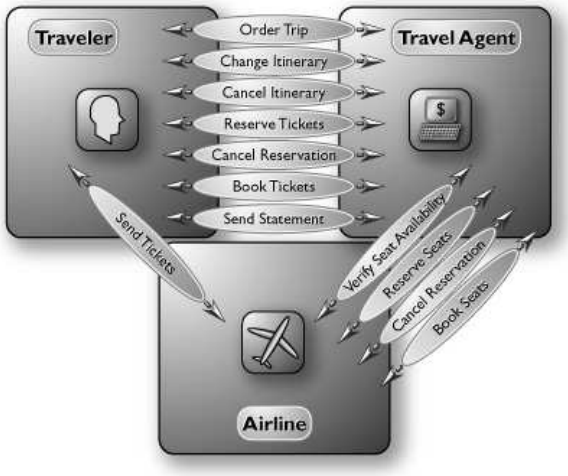

Fig. 1: Plan and Book Trip: Message Flow.

if a final confirmation has not been received before that day is over, the seats are released, and the Travel Agent is informed. Thus, the Traveller can now either finalize the reservation or cancel it. If he confirms the reservation (*Book Tickets*), the Travel Agent asks the Airline Reservation System to finally book the seats (*Book Seats*).

The high level flow of the messages exchanged within the global process (which is called *PlanAndBookTrip*) is shown in Fig. 1.

2.1 WSCDL Description

In Figure 2 we can see a detailed piece of the WSCDL document describing our case study. It describes a part of the relationship between the Airline and the Travel Agent. The interaction determines that the reservation is available for one day.

2.1.1 The travel Agent

In Figure 3 we can see a part of the travel agent specification. This particular piece defines an exception to handle the reservation timeout. Overall the model for the travel agent has the following elements:

- The main activities of the travel agent are represented by nested processes.
- The iterative processes are described by means of *while* activities.
- Exceptions capture the withdrawal of the trip request or the reservation request.

```
<interaction name="reservation&booking"
        channelVariable="travelAgentAirlineChannel"
        operation="reservation&booking"
        align="true"
        initiate="true" >
 <participate relationshipType="TravelAgentAirline"
        fromRole="TravelAgent" toRole="Airline" />
 <exchange name="reservation"
        informationType="reservation" action="request" >
  <send variable="tns:reservationOrderID" causeException="true" />
  <receive variable="tns:reservationAckID" causeException="true" />
 </exchange>
 <exchange name="booking" informationType="booking" action="respond" >
  <send variable="tns:bookingRequestID" causeException="true" />
  <receive variable="bookingAckID" causeException="true" />
 </exchange>
 <timeout time-to-complete="24:00" />
 <record name="bookingTimeout" when="timeout" causeException="true" />
  <source
   variable="AL:getVariable('tns:reservationOrderCancel', '', '')" />
  <target
   variable="TA:getVariable('tns:reservationOrderCancel', '', '')" />
 </record>
</interaction>
```

Fig. 2: A Part of the WS-CDL Specification

```
...
<context>
 <process name ="BookSeats" instantiation="other">
  <action name="bookSeats"
        role="tns:travelAgent"
        operation="tns:TAtoAirline/bookSeats">
  </action>
 </process>
 <exception>
  <onMessage>
   <action name="ReservationTimeOut"
        role="tns:TravelAgent"
        operation="tns:TAtoAirline/AcceptCancellation">
     <correlate
        correlation="defs:reservationCorrelation"/>
   </action>
   <action name="NotifyOfTimeOut"
        role="tns:TravelAgent"
        operation="tns:TAtoTraveller/NotifyofCancellation"/>
              <fault code="tns:reservationTimedOut"/>
  </onMessage>
  ...
  </exception>
  ...
  </context>
```

Fig. 3: A Part of the Travel Agent Specification

- The interface uses two different correlations, which identify the same conversation involving the travel agent with both the traveller and the airline reservation system.

2.1.2 Traveller

The main top-level process describing the traveller is declared with an "`instantiation=other`" attribute to describe the fact that the traveller is actually the one who starts the message exchange.

To model the possibility of cancelling the reservation or the book the tickets we use a new context with a new exception.

We use a correlation to ensure that both the travel agent and the airline reservation system know how to fulfill the correlation requirements exhibited by the traveller interface.

```
...
<sequence>
 <context>
  <exception>
   <onTimeout property ="tnsd:expireTime"
          type="duration"
          reference="tns:ReserveSeats@end">
     <compensate name="CompensateReservation"
     transaction="seatReservation"/>
   </onTimeout>
  </exception>
 </context>
  ...
</sequence>
```

Fig. 4: Part of the Travel Agent Specification

2.1.3 Airline Reservation System

The airline reservation system interface is modeled by an interface with two top-level processes, both with the "`instatiation=message`" attribute.

The reservation of seats for each leg is defined as a transaction defining a compensation activity which probably will withdraw the reservations for all seats.

Fig. 4 shows the part of the specification used to control the timeout.

3 Modelling with Timed Automata

For each component of a WS-CDL description we have the following correspondences in timed automata:

Role : They describe the behaviour of each class of party that we are using in the choreography. Thus, this definition matches with the definition of a automaton *template* in timed automata terminology.

Relation type : They define the communications between two roles, and the needed channels for these communications. In timed automata we assign a new channel for each one of these channels, which are the parameters of the templates that take part in the communication.

Participant type : They define the different parties that participate in the choreography. In timed au-

tomata they are *processes* or automata participating in the system.

Channel types : A channel is a point of collaboration between parties, together with the specification of how the information is exchanged. As said before, channels of WS-CDL correspond with channels of timed automata.

Variables : They are easily translated, as timed automata in UPPAAL support (bounded integer) variables, which are used to represent information.

At this point it is necessary to define the behaviour of each template. This behaviour is defined by using the information provided by the flow of choreographies. Choreographies are sets of workunits or sets of activities. Thus, activities and workunits are the basic components of the choreographies, and they capture the behavior of each component. Activities can be obtained as result of a composition of other activities, by using sequential, parallel or choice compositions. In terms of timed automata these operators can be easily translated:

- Sequential composition of activities is translated by concatenating the corresponding timed automata.
- Parallel activities are translated to the flat parallel composition of the corresponding timed automata.
- Choices are translated by adding a node into the automata which is connected with the initial nodes of the alternatives.

Finally, time restrictions are associated in WS-CDL with workunits and interaction activities. These time restrictions are introduced in timed automata by means of a local clock, guards and invariants. Therefore, when a workunit of an activity has a time restriction, we associate a guard to the edge that correspond to the initial point of this workunit in the corresponding timed automaton.

In Fig. 5 we can see the schematic presentation of the correspondence between WS-CDL and timed automata.

Now, we can see this correspondence for our example. When we model our case Study by using Timed Automata we have 3 automata, the automaton of the traveler, the automaton of travel agent and another one for the airline company.

In Fig. 6 we can see the timed automaton corresponding to the airline reservation system. In this

$$
\begin{array}{ll}
Role & \longrightarrow Template \\
RelationType & \longrightarrow Channel^{+} \\
ParticipantType & \longrightarrow Process^{+} \\
ChannelType & \longrightarrow Channel \\
Variables & \longrightarrow Variables \\
Choreography & \longrightarrow Choreography^{+} \mid \text{Activity} \\
Activity & \longrightarrow WorkUnit \mid \text{Sequence} \mid \\
 & \quad \text{Paralelism} \mid \text{Choice} \\
Sequence & \longrightarrow Activity^{+} \\
Paralelism & \longrightarrow Activity^{+} \\
Choice & \longrightarrow Activity^{+} \\
WorkUnit & \longrightarrow State\&Guard\&Invariant
\end{array}
$$

Fig. 5: Schematic view of the translation

automaton we use the clock x to control when the reservation expires. This clock is initialized when a reserved_seat is done.

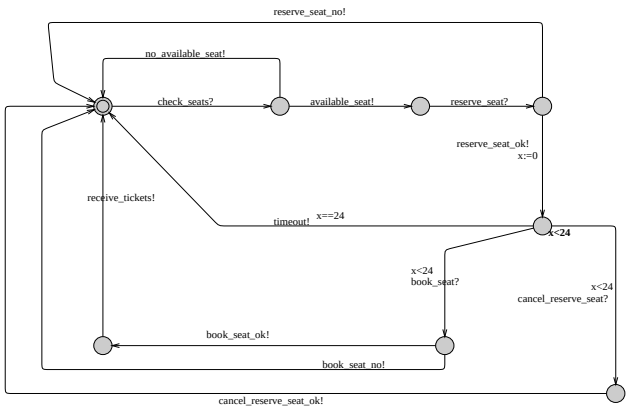

Fig. 6: Timed automaton for Airline Reservation System.

Fig. 7 shows the timed automaton of the Travel Agent and Fig. 8 present the automaton for the traveler.

4 Simulation and Verification

Once we have constructed the timed automata describing the system, we can use the UPPAAL tool to simulate and verify the system it.

Simulations can detect some failures in the system design, and thus, this can be the first step to verify the system behavior. Simulations are made by choosing different transitions and delays along the system evolution. At any moment during the simulation, you

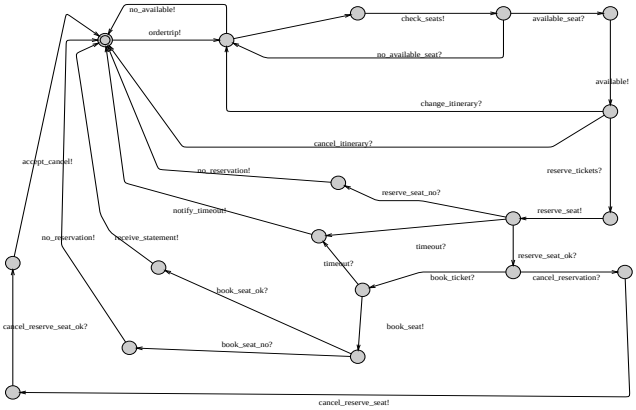

Fig. 7: Timed automata for Travel agent web service.

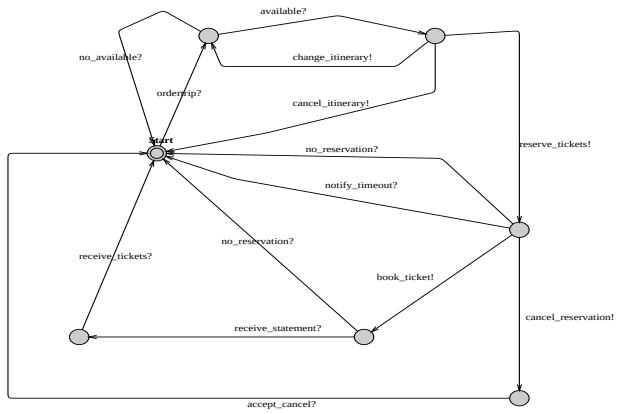

Fig. 8: Timed automata for traveler.

can see the variable values and the enabled transitions. Thus, you can choose the transition that you want to execute. Nevertheless, you can also select the random execution of transitions, and thus, the system evolves by executing transitions and delays which are selected randomly. We have some other options in the UPPAAL Simulator. For example, you can save simulations traces that can later be used to recover a specific execution trace. Actually, the simulation is quite flexible at this point, and you can back or forward in the sequence.

Then, with respect to our case study, our main goal in the verification phase is to check the correctness of the message flow and time-outs, taking into account the protocol definition. We have made a number of simulations, and for all of them the system has satisfied the expected behavior in terms of the message flow between the parties.

However, simulations do not guarantee in general the correctness of a system, as they cannot be complete, so we must also use formal verification techniques to complete the verification process of the system. But before starting the automatic verification, we must establish which are the properties that the model must fulfill. We have divided these properties into three classes: Safety, Liveness and Deadlocks. These properties are

Safety Properties allow us to check if our model satisfies some security restrictions. For example, if we have two trains that have to cross the same bridge, a security property is that both trains can not cross at the same time the bridge, this is described by a formula like: $\forall\Box\neg(Train1.crossing \wedge Train2.crossing)$ or $\neg\exists\Diamond(Train1.crossing \wedge Train2.crossing)$.

In our case study, the main Safety properties that our system must fulfill are the following:

- The TravelAgent always sends the itinerary on traveler's demand:

$$\forall\Box Traveler.Itinerary \Rightarrow TravelAgent.sendItinerary \quad (1)$$

- The TravelAgent always changes the itinerary on traveler's demand:

$$\forall\Box Traveler.ChangeItinerary \Rightarrow TravelAgent.PerformChange \quad (2)$$

- The TravelAgent always cancels the reservation on traveler's demand:

$$\forall\Box Traveler.CancelReservation \rightarrow (TravelAgent.CancelReservtRcv \wedge Airline.PerformCancel \wedge Airline.Clockx < 24) \quad (3)$$

- A reservation is only available 24 hours before performing the booking:

$$\forall\Box(TravelAgent.Booking \wedge Airline.ReceiveBoking \wedge Airline.ClockX <= 24) \quad (4)$$

- A Traveler always receives theirs tickets and statements after performing the payment:

$$\forall\Box Traveler.PaymentPerform \rightarrow (Traveler.Finish \wedge Airline.SnddTckt \wedge TravelAgent.SenddSttment) \quad (5)$$

Liveness Properties are used to check that our model can evolve in the right order. Returning to the train example, if a train approaches the bridge, some time later, the train could cross it: $Train.approach \rightarrow Train.crossed$.

In our case study we can consider quite a number of liveness properties. Then, we have just selected two of them, which are one of the most important:

- If a Traveler sends a trip demand, some time later, the TravelAgent will send the itineraries. Translating it into Temporal Logic we have:

$$Traveler.PlanOrder \longrightarrow TravelAgent.SendItinerary \quad (6)$$

- If a Traveler makes a book within 24 hours after the reservation, the Airline performs the booking. Translating it into Temporal Logic we have:

$$Traveler.BookOdr \wedge Airline.ClockX < 24 \longrightarrow Airline.PerformBook \quad (7)$$

Deadlocks are clear mistakes. We can check if our model can deadlock by the formula:

$$\forall\Box\neg Deadlock \quad (8)$$

5 Conclusions and Future Work

Web Services are widely used to implement a great variety of applications on Internet. Many of them have time restrictions, and then, it becomes important to verify that they will comply with these restrictions, even before these systems are implemented. Then, this paper shows how formal techniques can be used to this purpose, and particularly timed automata. We have seen that the specifications of Web Services applications can be translated into timed automata, and we have illustrated this translation by means of an example, and then, we have seen how these automata can be simulated and how we can verify the main properties of the system by using these automata.

References:

[1] H. Davulcu, M. Kifer, and I. V. Ramakrishnan. CTR-S: A Logic for Specifying Contracts in Semantic Web Services. In *Proceedings of WWW2004*, pages 144–153, May 2004.

[2] A. Paschke, J. Dietrich, and K. Kuhla. A Logic Based SLA Management Framework. In *4th Semantic Web Conference (ISWC 2005)*, 2005.

[3] G. Governatori. Representing business contracts in RuleML. *International Journal of Cooperative Information Systems*, 14:181–216, 2005.

[4] E. S. C. Molina-Jimenez, S. Shrivastava and J. Warne. Run-time Monitoring and Enforcement of Electronic Contracts. *Electronic Commerce Research and Applications*, 3(2), 2004.

[5] A. Daskalopulu. Model Checking Contractual Protocols. In L. Breuker and Winkels, editors, *Legal Knowledge and Information Systems, JURIX 2000: The 13th Annual Conference*, Frontiers in Artificial Intelligence and Applications Series, pages 35–47. IOS Press, 2000.

[6] Nickolas Kavantzas et al. *Web Service Choreography Description Language (WSCDL) 1.0*. In http://www.w3.org/TR/ws-cdl-10/

[7] R. Alur and D. Dill, *Automata for modeling real–time systems*, In Proceedings of the 17th International Colloquium on Automata, Languages and Programming, volume 443, Editors. Springer–Verlag, 1990.

[8] H. Foster, S. Uchitel, J. Magee, J. Kramer, *Leveraging Eclipse for Integrated Model-Based Engineering of Web Service Compositions*, In ETX2005 Workshop at OOPSLA05, San Diego, CA, October 2005.

[9] Edmund M. Clarke and Jr. and Orna Grumberg and Doron A. Peled, *Model Checking*, MIT Press, 1999.

[10] K. Larsen and P. Pettersson and Wang Yi, UPPAAL *in a Nutshell*, Int. Journal on Software Tools for Technology Transfer, Editors. Springer–Verlag vol.1, 1997.

[11] G. Diaz, F. Cuartero, V. Valero and F. Pelayo, *Automatic Verification of the TLS Handshake Protocol*, In proceedings of the 2004 ACM Symposium on Applied Computing.

[12] G. Diaz, K.G. Larsen, J. Pardo, F. Cuartero and V. Valero, *An approach to handle Real Time and Probabilistic behaviors in e-commerce: Validating the SET Protocol*, In proceedings of the 2005 ACM Symposium on Applied Computing.

[13] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis and S. Yovine, *Kronos: A model-checking tool for real-time systems*, In Proc. 1998 Computer-Aided Verification, CAV'98, Vancouver, Canada, June 1998. Lecture Notes in Computer Science 1427, Springer-Verlag.