Chapter 7

Simulating Semantic Web Services with LSCs and Play-Engine

As introduced in Chapter 1, the full potential of the Web is realized when not only static information, but also dynamic Web services, are processable by software agents.

Web Services provide a standard way of interoperation between applications that may be running on a variety of platforms. The interoperability is achieved by the development of employment of a set of XML-based open standard protocols/languages such as WSDL [14], SOAP [110] and UDDI [99]. Web services encoded in such protocols can be autonomously understood by applications.

Although the above languages are still in evolution, it has been recognized that there is a growing need for semantically richer specification languages. Such *semantical* specifications can further automate various activities of the life cycle of a Web service, such as service invocation, selection, composition, negotiation, etc. For these reasons the Semantic Web Services ontology was developed.

The Semantic Web Services ontology, called OWL-S, is an ontology in OWL DL language. As introduced previously, it contains essential mark-ups for describing a Semantic Web service. Such markups can be categorized into three parts: service grounding, service profile and service model. The details can be found in Chapter 2.2. The service model component describes the how the service works, detailing its inputs, outputs, preconditions, effects, control flow, etc. Hence, it is essential to the selection and invocation of a service. It is important to ensure the correctness of such models as erroneous service descriptions will give rise to invocation of wrong services, with wrong parameters, resulting in undesired outcome.

In this chapter, we demonstrate how to encode Semantic Web service models as Live Sequence Charts (LSCs) and how to simulate them using Play-Engine.

The chapter is divided into four sections. Section 7.1 is devoted to an introduction to the LSCs and Play-Engine, the visualization and simulation tool support for LSC. In Section 7.2, we introduce how OWL-S ontologies are transformed into LSCs. In Section 7.3, we demonstrate the simulation process through a case study of an online holiday booking system. Finally, Section 7.4 summarizes the chapter.

7.1 LSCs & Play-Engine

Live Sequence Charts (LSCs) [18] are a powerful visual formalism which serves as an enriched requirements specification language. LSCs are a broad extension of the classic Message Sequence Charts (MSCs [53]). They capture communicating scenarios between system components rigorously. LSCs distinguish scenarios that must happen from scenarios that may happen, conditions that must be fulfilled from conditions that may be fulfilled, etc.

There are two kinds of charts in LSCs: existential charts and universal charts. Existential charts are mainly used to describe possible interactions between participants in early stages of system design. At a later stage, knowledge becomes available about when a system run has progressed far enough for a specific usage of the system to become relevant. Universal charts are then used to specify behaviors that should always be exhibited. A universal chart may be preceded by a pre-chart, which serves as the activation condition for executing the main chart. Whenever a communication sequence matches a pre-chart, the system must proceed as specified by the main chart. A chart typically consists of multiple instances, which are represented as vertical lines. Along with each line, there are a finite number of locations (i.e., the joint points of instances and messages). A location carries the temperature annotation for progress within an instance. Message passing between instances is represented as horizontal lines. Cold conditions are used to assistant specifying complex control structures like guarded-choice, do-while. Hot conditions are asserted to assure critical properties at certain point of execution. Typically, a system is described by a set of LSCs, both universal charts and existential charts. LSCs support advanced MSC features like co-region, hierarchy, etc. For details on features of LSCs, refer to [37]. LSCs are far more expressive than MSCs, which makes them capable of expressing complicated inter-objects system requirements.

An interaction-based model specifies the desired inter-object relationships before a system is actually constructed. It is beneficial if the model can be simulated and tested so as to detect inconsistencies and under-specification. One of the significance of LSCs is that descriptions in the LSC language can be executed by Play-Engine [38] without implementing the underlying object system. Play-Engine is a tool recently developed to support an approach to the specification, validation, analysis and execution of LSCs, called "play-in" and "play-out". Behaviors are "played in" directly from the system's user interface, and as this is being done the Play-Engine continuously constructs LSCs. Later, behaviors can be "played out" freely from the user interface, and the tool executes the LSCs directly, thus driving the system's behaviors. When "playing out", Play-Engine computes a "maximal response" to a user-provided event, called a super-step. During the computation of a super-step, hot conditions are evaluated. If any hot condition evaluates to false, a violation is caught. Otherwise, simulation continues with the user provided events. This way, users may detect undesired behaviors allowed by the specification early in the development. The basic play-out engine arbitrarily explores a single super-step, hence possibly running into problems. The smart play-out approach uses model checking to compute a valid super-step if it exists. Alternatively, test cases may be supplied by the users as existential charts so that Play-Engine may guide the system accordingly to verify that a scenario of interactions between the user and system is possible.

7.2 Modeling OWL-S with LSCs

7.2.1 Basics

The work in this chapter is concentrated on the process model of OWL-S and we abstract away the service profile and grounding details. The key idea of using LSCs to visualize and simulate the OWL-S process models is to use an LSC universal chart to capture a process model. In other words, each process is viewed as describing a possible communicating scenario between a service-using agent and the service-providing agent. For each process model, we assume there is a pre-service request from the service-using agent to the service-providing agent that identifies the service to perform, which corresponds to the service grounding phase that we ignore in this work. For instance, the *request*() message in Figure 7.2 is a pre-service request from a *HolidayBookingAgent* to a *BdgtChker*. Once a pre-service request is exchanged between the service-using agent and the service-providing agent, subsequent interactions follow precisely as defined in the service definition (the process model).

In OWL-S, processes are modeled as OWL classes and they are sub classes of one of the three mutually disjoint OWL classes: *AtomicProcess*, *SimpleProcess* and *CompositeProcess*.

Processes can have inputs, outputs, preconditions, effects (IOPEs) and results, which are also defined as OWL classes. A result bundles (conditioned) effects and outputs.

Besides defining these classes, the OWL-S ontology also defines a number of object properties that defines the IOPEs of a process. The following list briefly explains these properties.

- *hasInput*: It specifies one of the inputs of the service.
- *hasLocal*: It specifies one of the local parameters. Local parameters are only used in atomic processes.
- *hasOutput*: It specifies one of the outputs of the service.
- *hasPrecondition*: It specifies one of the preconditions of the service. Preconditions are evaluated with respect to the client environment before the process is invoked.
- *hasResult*: It specified one of the *Results* of the service. Results can be associated with post-conditions by the property *inCondition*. Result conditions are effectively meant to be 'evaluated' in the server context after the process has executed. The outputs and effects of a result can only occur if its conditions are evaluated to true.

  Post-condition of the *inCondition* properties in *hasResult* are conjoined and identified with a shared hot condition at the end of the chart so that if the post-condition is violated, an error is raised by Play-Engine. The *withOutput* properties are then identified with communications after the hot condition.

7.2.2 Processes

An atomic process corresponds to the actions that a service can perform by engaging it in a single interaction, i.e., a one-step service that expects a bundle of inputs and produces a bundle of outputs. An atomic process is a "black box" representation; that is, no description is given of how the process works (apart from IOPEs).

Basically, a service defined by an atomic process is translated to an LSC universal chart preceded by a pre-chart containing only the pre-service request. An atomic process has always two participants, i.e., a service-using agent and a service-providing agent if the participants are skipped in the OWL-S ontology. Otherwise, participants in an ontology are translated to instances in the chart. According to [95], "inputs and outputs specify the data transformation produced by the process", hence they are identified with communication between different participants in the main chart. If a process has a precondition, it cannot be performed successfully unless the precondition is true. Precondition of a service is, therefore, identified with a shared cold condition (among all participants) at the very beginning of the main chart. Thus, if the condition is violated, the chart terminates and hence the process (service) is not performed.

The data bindings are analyzed to identify the correspondence between different inputs and outputs and local variables (if there are). Besides, built-in functions in the process models are translated to external functions in LSC (Play-Engine) and local variables are identified with variables associated with the instances in the chart.

Composite processes are composed of sub-processes, and specify constraints on the ordering and conditional execution of these sub-processes. These constraints are captured by the *composedOf* property. Composite processes are constructed using control constructs and references to processes called *Performs*. These are analogous to function calls in procedural language function bodies. *Perform* itself is a kind of control construct specifying where the client should invoke a process provided by some server. *Perform* may be references to atomic or other composite processes. *Performs* are composed using other control constructs. The minimal initial set includes *Sequence*, *Split*, *Split+Join*, *Any-Order*, *Condition*, *If-Then-Else*, *Iterate*, *Repeat-While* and *Repeat-Until*. We summarize the list of control constructs in Table 7.1 (according to OWL-S 1.1).

Table 7.1: A Partial Summary of the OWL-S constructs

OWL-S Constructs	**Description**
Sequence	Executes a list of processes in order.
Split	Executes a bag of processes concurrently.
Split+Join	Executes a bag of processes concurrently with barrier synchronization.
Any-Order	Execute a bag of processes in any order but not concurrently.
Choice	Chooses between alternatives and executes.
If-Then-Else	Tests the *if-condition*. If *true* executes the "Then" branch, if *false* executes the "Else" branch.
iterate	Serves as the common superclass of *Repeat-While* and *Repeat-Until* and potentially other specific iteration constructs.
Repeat-While	Iterates execution of a bag of processes until the *while* Condition becomes true.
Repeat-Until	Iterates execution of a bag of processes until the *until* Condition becomes true.
timeout	Interval of time allowed for completion of the process component (relative to the start of process component execution).

In the following, we discuss how composite services are systematically transformed to LSCs. We present the transformation in the following as transformation rules for each and every control construct in Table 7.1.

- *Sequence*: It is naturally translated to sequential communication along the vertical lines in a chart. If a sub-process itself is composed by other processes, the sub-process is transformed to a sub-chart or a pre-service request in case the sub-process is reused in other processes. Variables in the output bindings are parameterized with the message so that they are unified with the variables in the invoked processes.
- *Split*: Because no specification about waiting or synchronization is made among the bag of process components, processes in *Split* correspond to multiple pre-service requests grouped as a co-region so that the ordering of the execution of the components are not constrained. Each pre-service request will in turn activate an LSC modeling the corresponding service.
- *Split+Join*: Because of the possible barrier synchronization, it is transformed to LSCs similarly as *Split* with additional 0-buffered communication corresponding to the barrier synchronization. The 0-buffered communication events are shared by all LSCs modeling the invoked services. Therefore, the synchronization is made among all sub-processes. Moreover, the location where the co-region is set to be hot so that completion of all components are guaranteed.
- *Any-Order*: All components of an *Any-Order* control construct must be executed, but not concurrently. This requires that no execution of any two processes can overlap. This is transformed to LSCs exactly as *Split* except all locations in LSCs corresponding to the components are set to be hot so that completion of all components are guaranteed.
- *Choice*: This corresponds to the *Select-Case* construct in LSCs. Thus, a choice in OWL-S is transformed to a *Select-Case* sub-chart with equally distributed possibility.
- *If-Then-Else*: The exact same construct *if-then-else* is available in LSCs. The

If-condition and *Else-condition* are mapped to cold conditions in the respective sub-chart. The only problem is to syntax-rewrite the logical expression used in OWL-S (represented in SWRL [48], DRS[1] or KIF[2]) properly to logical expression in LSCs.

- *Repeat-While and Repeat-Until*: Both these two constructs are sub classes of the *abstract* control construct *Iterate*, whereas the former is transformed to a looping sub-chart in LSCs with a shared cold condition (corresponding to the condition in the service definition) at the end of the sub-chart and the latter is transformed to a looping sub-chart in LSCs with a cold condition (corresponding to the negation of the condition in the service definition) at the end of the sub-chart.
- *timeout*: *timeout* is defined as an object property on the above control constructs, each of which can have at most 1 such timeout instance. It is mapped to a timer set event followed by a timeout event in LSCs containing the respective process components.

The transformation rules for composite processes are applied inductively. One of the difficulties of using LSCs to simulate the OWL-S process models is to perform correct data binding and data computation. We assume that a simple underlying data and functional model of the system is supplied by the users, i.e. the underlying system variables and the implementation of the external functions and so on. To simulate the set of process models interactively, we may build a simple user interface to trigger environmental events manually. A simple user-interface is built with a button for triggering every process model. Play-Engine supports building such user-interface with Visual Basic, and “playing-out” the corresponding LSCs according the

[1]cf. `http://www.daml.org/services/owl-s/1.0/conditions.html`
[2]cf. `http://logic.stanford.edu/kif/dpans.html`

user interaction through the interface.

7.3 Case Study

This section illustrates the approach with an example of an online holiday booking system.

7.3.1 System scenario

The holiday booking system is a Web portal offering access to information about air tickets and hotels. This Web portal provides automated air ticket and hotel booking services to users who are planning their holidays.

In the course of operation, the customer submits a request, which includes the information about the destination, travelling time and maximum budget, to the holiday booking agent. Upon receiving the request, the holiday booking agent tries to find the most suitable air ticket and hotel based on information in the customer's preferences, which have been obtained from his online, OWL-encoded profile. The preferences may include the preferred airlines, hotels, etc. Following that, the holiday booking agent calculates if the total cost overruns the budget limit. If the total cost is more than customer's budget, the holiday booking agent tries to find another cheaper hotel or ticket. If there is no ticket and hotel combination that can be found within the budget, the customer will be notified. Otherwise the booking agent shows the information about the matched ticket and hotel to the customer. If the customer is satisfied, he/she submits his/her credit card information to the holiday booking agent. The holiday booking agent asks a third-part credit checking agent to check if the card is valid with sufficient credit. If it is, the booking will be made.

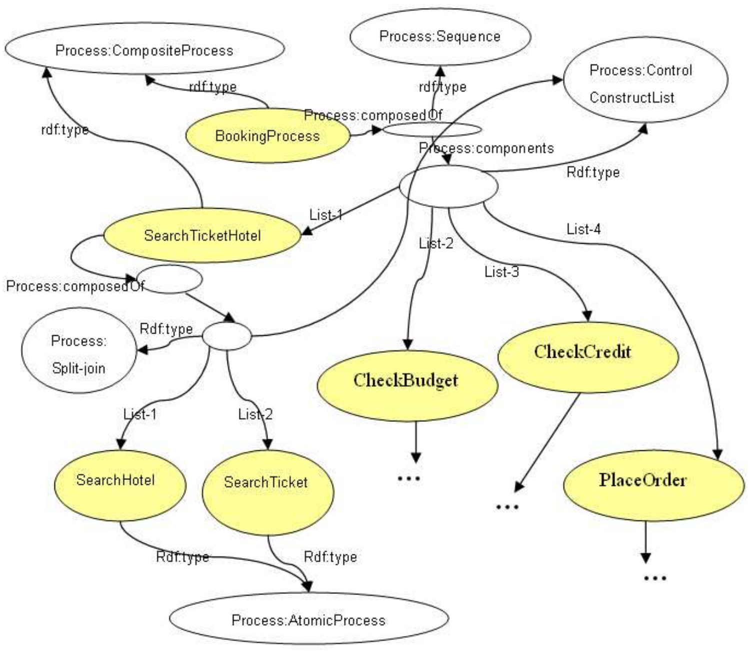

Figure 7.1: Holiday booking System

Figure 7.1 is an RDF graph of the service model ontology. It shows part of the OWL-S process model for the holiday booking agent[3]. The holiday booking service has a composite process *BookingProcess* which sequentially performs four sub-processes – *SearchTicketHotel*, *CheckBudget*, *CheckCredit* and *PlaceOrder*. *SearchTicketHotel* is a composite process as well, which performs two atomic process, *SearchHotel* and *SearchTicket*, in parallel. The complete OWL-S process model can be found at http://www.comp.nus.edu.sg/~liyf/booking.xml.

Being part of our case study, the following is the process model of an atomic OWL-S service ontology that checks whether the current air ticket and hotel prices are within user budget, given as inputs the air ticket price (variable X1), hotel accommodation cost (variable X2) and the user's budget (variable X3)[4]. As output, this atomic service

[3]The diagram has been slightly revised for presentation purpose.

[4]These variables are represented as `budget_ticket_Cost`, `budget_hotel_Cost` and

returns `true` for variable `Check_Budget_result` if $\texttt{X3} \leq \texttt{X1} + \texttt{X2}$, and false otherwise. For atomic processes, the inputs must come from the service-using agent.

```
<process:AtomicProcess rdf:ID="CheckBudget">
  <process:hasInput><process:Input rdf:ID="budget_hotel_Cost">
    <process:parameterType rdf:datatype="&xsd;#nonNegativeInteger"/>
  </process:Input></process:hasInput>
  <process:hasInput><process:Input rdf:ID="budget_ticket_Cost">
    <process:parameterType rdf:datatype="&xsd;#nonNegativeInteger"/>
  </process:Input></process:hasInput>
  <process:hasInput><process:Input rdf:ID="budget_total_Cost">
    <process:parameterType rdf:datatype="&xsd;#nonNegativeInteger"/>
  </process:Input></process:hasInput>
  <process:hasOutput><process:Output rdf:ID="Check_Budget_result">
    <process:parameterType rdf:datatype="&xsd;#anyURI">&xsd;#boolean
  </process:parameterType></process:Output></process:hasOutput>
  <process:hasResult>
    <process:Result rdf:ID="Within_budget">
      <process:withOutput>
        <process:OutputBinding>
          <process:toParam rdf:resource="#Check_Budget_result"/>
          <process:valueData rdf:datatype="&xsd;#boolean">true
          </process:valueData></process:OutputBinding></process:withOutput>
      <process:inCondition>
        <expr:KIF-Condition>
          <expr:expressionBody>
              (>= ?budget_total_Cost
                  (+ ?budget_ticket_Cost ?budget_hotel_Cost))
          </expr:expressionBody>
        </expr:KIF-Condition>
      </process:inCondition>
    </process:Result>
  </process:hasResult>
  <process:hasResult>
    <process:Result rdf:ID="beyond_budget">
      ...
    </process:Result>
  </process:hasResult>
</process:AtomicProcess>
```

Figure 7.2 shows an LSC universal chart capturing the necessary interactions between a service-using agent and a budget-checking agent cooperating in the above

`budget_total_Cost` in the ontology, respectively.

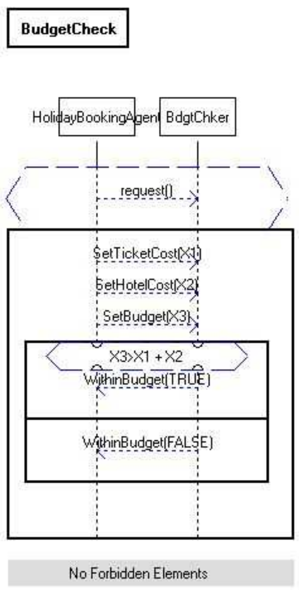

Figure 7.2: LSC Example: Budget checking

atomic service. Once the service-using agent requests the service *CheckBudget* (after determining whether the service meets its needs by exploring the service profile), necessary information like `budget_ticket_Cost` and `budget_hotel_Cost` is supplied by the service-using agent. The budget-checking agent replies with true, if the budget is at least as much as the sum of the air ticket and hotel prices, and false otherwise.

7.3.2 Simulation

Figure 7.3 shows in Play-Engine part of the LSC of the *HolidayBooking* process model. Given a set of inputs including departure and destination cities, outbound and inbound dates, budgets, etc., the service searches for valid air tickets and hotels. Finally if such flights and hotel accommodation are available, it proceeds to book the flight and room.

Our simulation begins with building a simple Graphical User Interface (GUI) for

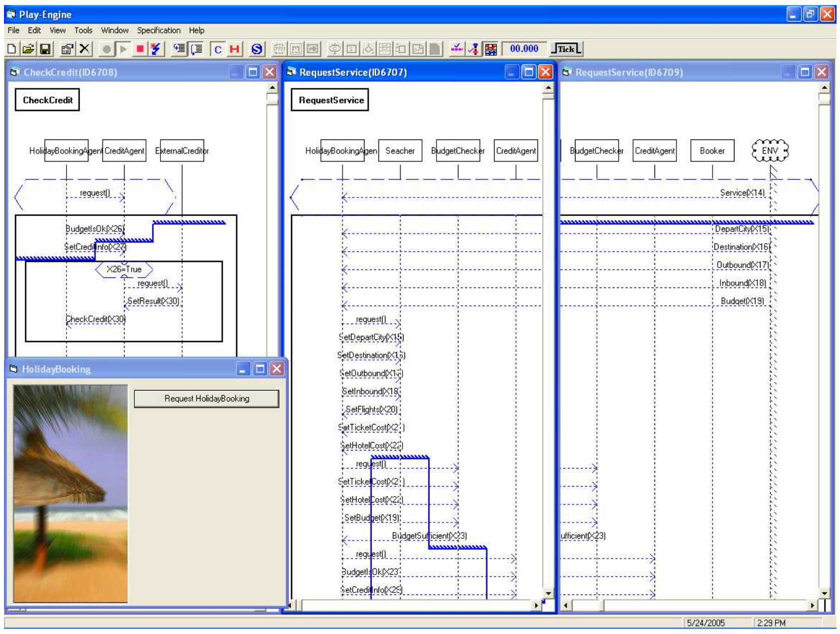

Figure 7.3: Simulation Screen Shot

interactively introducing external events. A systematic approach is to build one GUI component for each user-accessible Web service. In our example, only one Web service is accessible to service-using agents, namely *HolidayBooking*. The simple GUI is shown in the left bottom corner of Figure 7.3. Play-Engine allows user-defined variables and external function through ActiveX DLLs. For the purpose of simulation before actual implementation, an abstract "implementation" capturing only necessary details of the system is sufficient. However, if the underlying data and functional system is implemented using techniques compatible with ActiveX DLLs, e.g. ASP, .NET, Play-Engine may import the actual implementation of the underlying system and perform the simulation.

From our experiences, symbolic messages and instances are very helpful for capturing the OWL-S process models compactly. After building the LSC model, a user may interactively play out the system by initiating an (or a series of) external event and check how the system proceeds step-by-step. Assertion can be inserted freely by introducing hot conditions in the LSCs. During simulation, a violation of the hot condition will be caught by Play-Engine. This way, inconsistency and under-specification is detected intuitively. In case an external process (to be offered by third party) is assumed, the user may specify the possible output of the process manually or Play-Engine would use model-checking techniques to automatically find a valid value (if the variables have finite domain). In our example, during simulation, windows pop up for the user to specify the ticket price and the hotel price. Alternatively, a user may build a test case of the system as an existential chart (with assertions) and let Play-Engine do the guided play-out according the existential chart.

In Figure 7.3, the *HolidayBooking* process is invoked by two different service-using agents. Hence, two copies of the chart *HolidayBooking* (according to the *HolidayBooking* process) are monitored. With simulation run of this scenario, where a number of service-using agent are using the ticket-booking service, we gain confidence that the same shared resource (e.g. ticket vacancy) is accessed exclusively.

7.4 Chapter Summary

In this chapter, we propose to use LSCs and Play-Engine to visualize and simulate OWL-S process models. The significance and novel aspects can be summarized as follows. Firstly, by transforming an OWL-S service model ontology into an LSC, service developer can design the services in a more visual and intuitive manner. In XML format, the LSCs can be easily transformed back to OWL-S. Secondly, we may simulate the interactions without implementing the Web service (exploring the service grounding), and be able to gain confidence of the service models. The key point of this approach is that a Web service can be naturally viewed as a desired usage of the web agent, i.e., a scenario of the interaction between the service-using agent the service-providing agent. Thirdly, as Play-Engine supports dynamic linked libraries such as COM and ActiveX Controls, Web services written in these libraries can be more easily transformed to LSCs, from which the OWL-S service model may be derived. Hence, our approach also facilitates the integration of Web services with OWL-S. Moreover, we presented a travel booking case study to demonstrate our approach.

There are a number of future work directions that we deem as worthwhile to pursue. First of all, it is necessary to develop programs to automatically construct LSCs from the OWL-S process models to make this approach more practical. Recently an OWL-S editor has been developed[5] as a plug-in for the Protégé OWL Editor [57]. It will be valuable for OWL-S developers if they can obtain feedback, in terms of simulation results, from Play-Engine simulations directly to the editor. Hence, such a deep linking between Play-Engine and the OWL-S editor is desirable. Besides LSC and Play-Engine, formal languages such as CSP [42] can also be considered to represent OWL-S ontologies and their tool support, such as the FDR [83] or SPIN [43] model checkers, may also be used to perform verification tasks. They are part of the future research plan that will be detailed in the next chapter.

We foresee that Web Services will be a new and fruitful application domain of Software Engineering (SE) methods and tools. Our approach, along with other approaches on applying SE methods to the Web domain, offers both experience and possible tool supports for developing Web services languages and techniques.

[5]cf. http://owlseditor.semwebcentral.org/index.shtml