Visualizing and Simulating Semantic Web Services Ontologies

Jun Sun[1] Yuan Fang Li[1] Hai Wang[2] Jing Sun[3]

[1] School of Computing
National University of Singapore
{sunj,liyf}@comp.nus.edu.sg
[2] Department of Computer Science
University of Manchester
hai.wang@cs.man.ac.uk
[3] Department of Computer Science
The University of Auckland
j.sun@cs.auckland.ac.nz

Abstract. The development of Web Services has transformed the World Wide Web into a more application-aware information portal. The various standards ensure that Web Services are interpretable and extensible, opening up possibilities for simple services to be combined to build complex ones. The Semantic Web presents a new mechanism for users and software agents to discover, describe, invoke, compose and monitor Web services. For these purposes the Semantic Web Services (OWL-S) ontologies have been developed to provide vocabularies to describe Web Services in a precise and machine-understandable way. It is necessary to ensure the ontological descriptions of the services capture the intended meaning as erroneous description may cause invocation of wrong services, with wrong parameters, resulting in undesired outcome. In this paper, we propose to apply software engineering method and tools to visualize, simulate and verify OWL-S process models. Namely, Live Sequence Charts (LSCs) is used to model services, capturing the inner workings of services, and its tool support Play-Engine is used to perform automated visualization, simulation and checking.
Keywords: Semantic Web Services, OWL-S, LSC, Play-Engine

1 Introduction

The World Wide Web has evolved from a static information repository to a current dynamic distributed information sharing and processing source. Web Services [2, 3] are one of the latest endeavors in this evolution. Together with layers of XML-based open standards [19, 18, 3], Web Services provide a framework for automated service advertisement, discovery, invocation, composition & inter-operation and execution monitoring. Web applications can be dynamically discovered, invoked and simple services can be composed to build more complex ones.

The Semantic Web [1] is another frontier of the Web development that is believed by many as the future of the Web, in which software agents can cooperate to accomplish tasks without human supervision. Web resources are given well-defined meaning so that they are readily available to human users as well as machines to understand and process. Resources on the Web are expressed in terms of ontologies, which define concepts and relationships of a particular domain. Based on description logics, OWL (Web Ontology Language) has been published by W3C as a Proposed Recommendation.

A Semantic Web Services ontology, the OWL-S [17], is currently being developed to *semantically* specify Web services. Expressed in OWL, it is a meta-ontology aimed at supplying the service producers/consumers with a core set of machine-interpretable vocabularies for precisely describing the properties and capabilities of Web services. It can be foreseen that the blending of OWL-S and various Web Services standards will present a more automated, effective approach to developing, deploying and utilizing Web services. As OWL-S ontologies define what a service (sometimes referred to as a process) does, how it works and what are its inputs, outputs, preconditions and effects (IOPEs), it is necessary that they capture the correct information. Erroneous definition of preconditions, for example, may make a service invoked when it should not be. Hence, tool support, especially reasoning and simulation tool support is highly desirable for Semantic Web Services developers.

A number of reasoning engines have been developed for ontology languages RDF, DAML+OIL and OWL, such as FaCT [11] and RACER [5]. All these tools are concentrated on deducing subsumption relationship (deducing whether one OWL class is a "sub class" of another class) and checking consistency of static Semantic Web ontologies. We foresee that, since OWL-S emphasize on service description, forthcoming tool support ought to efficiently capture the dynamic aspects of services.

In this paper, we propose to use the software engineering language Live Sequence Charts (LSCs) [4] and its tool support Play-Engine [8] to visualize and simulate OWL-S process model ontologies, which capture the essential information about how a service is to be invoked, executed and the expected result and outputs. LSCs are a broad extension of the classic Message Sequence Charts (MSCs [13]). 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. Together with various high-level operators like bounded loop, if-then-else, LSCs may well be used to specify complicated inter-object system requirements. One of the novel aspects of LSCs is that they allow a "play-in/play-out" approach to simulate and verify the requirements without implementing the underlying object systems [7], which is realized in Play-Engine. It allows users to interactively introduce a set of LSCs as behavioral requirements and automatically drive the execution of the requirements by employing formal verification techniques.

The essential idea of capturing OWL-S process models using LSCs is that a OWL-S process model may be perfectly viewed as describing a scenario of the interactions between a service-using agent and the service-providing agent. The benefits of modeling OWL-S process models as LSCs are many-fold:

– Allowing service designers to enjoy the visual power of LSCs by visually designing OWL-S process models in Play-Engine. As LSCs are expressed in XML, they can be easily transformed back to OWL-S ontologies when a service designer is satisfied with the simulation runs in Play-Engine.
– Using Play-Engine to simulate services without implementing them. By smart playing-out [6] OWL-S process models in Play-Engine, unwanted scenarios may be discovered early in the design stage.
– Tapping Play-Engine's ability of interacting with dynamic linked libraries (.dlls) such as COM, COM+, ActiveX Controls, service designers can more easily write LSCs specifications by directly calling functions from Web services defined as these libraries, therefore integrate existing Web services with OWL-S process models.

Moreover, we believe that mature development of the synthesis and verification techniques of LSCs and MSCs offers helpful guidance on designing and verifying Web Services.

The rest of the paper is organized as follows. In Section 2, an overview of Web Services, the Semantic Web, ontology languages RDF, OWL and OWL-S, the language LSC and tool Play-Engine is briefly presented. In Section 3, we present the transformation rules from OWL-S process model ontology to LSCs. The air ticket search and booking case study is introduced in Section 4. It is the running example of the paper. In this section, we also demonstrate how Play-Engine can be used to run automatic simulation of services and verify properties dynamically. Finally, Section 5 concludes the paper and discusses future work directions.

2 Overview

This section is devoted to a brief introduction of Semantics Web and Web Services, LSCs and Play-Engine. Interested readers are referred to [17] and [8] for detailed features of OWL Web Services and LSCs, respectively.

2.1 Semantic Web & Web Services

Web Services is a W3C coordinated effort to define a set of open and industry-supported specifications to provide a standard way of coordination between different software applications in a variety of environments. A Web service is defined as "a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards" [2].

The various specifications in the Web services domain are all based on XML, making information processing and interchange easier. However, as XML Schema only defines the syntax of a document, it is hard for software agents to understand the *semantics* of a Web service described using these specifications. A language that is both syntactically well-formed and semantical is therefore desirable.

The Semantic Web [1] is an envisioned extension of the current Web, in which resources are given machine-understandable, unambiguous meaning so that software agents can cooperate to accomplish complex tasks without human supervision. Resources in Semantic Web are marked up using ontologies, defining concepts of and relationships between resources. Ontology languages give basic vocabularies for expressing ontologies. The Web Ontology Language (OWL) [15] is the de-facto ontology language. Given a particular domain, OWL uses *classes* to represent abstract knowledge, use *properties* to relate different classes and use *individuals* to represent concrete entities that belong to various classes. It lays the foundation on which other ontologies can build.

OWL-S is an OWL-based Web service ontology, which supplies Web service producers/consumers with a core set of markup language constructs for describing the properties and capabilities of their Web services in an unambiguous, computer-interpretable form. OWL-S was expected to enable the tasks of 'automatic Web service discovery', 'automatic Web service invocation' and 'automatic Web service composition and inter-operation'. OWL-S consists of three essential types of knowledge about a service: the profile, the process model and the grounding. Figure 1 shows the high-level architecture of an OWL-S ontology. A ServiceProfile tells what the service does. It is the primary construct by which a service is advertised, discovered and selected. The ServiceGrounding tells how the service is used. It specifies how an agent can access a service by specifying, for example, communication protocol, message format, port numbers, etc.. The primary concern of our work in this paper is the OWL-S ServiceModel (also called process model), which tells how the service works. Thus, the class Service is described By a ServiceModel. It includes information about the service's inputs, outputs, preconditions and effects. It also shows the component processes of a complex process and how the control flows between the components.

The OWL-S process model is intended to provide a basis for specifying the behavior of a wide array of services. There are two chief components of an OWL-S process model – the process, and process control model. The process describes a Web Service in terms of its input, output, precondition, effects and, where appropriate, its component subprocess. The process model enables planning, composition and agent/service inter-operation. The process control model – which describes the control flow of a composite process and shows which of various inputs of the composite process are accepted by which of its sub-processes – allows agents to monitor the execution of a service request. The constructs to specify the control flow within a process model include Sequence, Split, Split+Join, If-

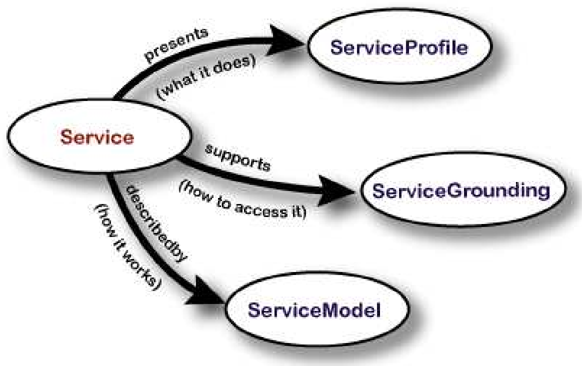

Fig. 1. Architecture of OWL-S Ontology

Then-Else, Repeat-While and Repeat-Until. The full list of control constructs in OWL-S and its semantics can be found in the latest version of OWL-S [17].

2.2 LSC & Play-Engine

LSCs are a powerful visual formalism which serves as an enriched requirements specification language. There are two kinds of charts in LSCs. 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 and etc. For details on features of LSCs, refer to [7]. 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 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". Behavior is "played in" directly from the system's user interface, and as this is being done the Play-Engine continuously constructs LSCs. Later, behavior can be "played out" freely from the user interface, and the tool executes the LSCs directly, thus driving the system's behavior. When "playing out", Play-Engine computes a "maximal response" to a user-provided event, called a super-step. During the computing 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 case may be supplied by the users as existential charts so that Play-Engine may guide the system accordingly to verify a scenario of interactions between the user and system is possible.

3 Modeling OWL-S with LSCs

In this section, we concentrate on the process model of OWL-S and abstract away the service grounding details. The key idea of using LSCs to visualize and simulate the OWL-S process models is to use an LSC universal chart capturing 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 3 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).

The classes of processes of a OWL-S ontology are categorized into three groups: atomic, composite and simple. 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 inputs, outputs, preconditions, and effects). The following are a list of process features of atomic processes.

- *process.hasInput*: It specifies one of the inputs of the service.
- *process.hasLocal*: It specifies one of the local parameters. Local parameters are only used in atomic processes.
- *process.hasOutput*: It specifies one of the outputs of the service.

- *process.hasPrecondition*: It specifies one of the pre-conditions of the service. Preconditions are evaluated with respect to the client environment before the process is invoked.
- *process.hasResult*: It specified one of the effects of the service. Result conditions are effectively meant to be 'evaluated' in the server context after the process has executed.

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. service-using agent and 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 [17], "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. Pre-condition 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. Post-condition of the *inCondition* properties in *process.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.

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. *PERFORM* 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 1 (according to OWL-S 1.1).

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 1.

- *Sequence*: It is naturally translated to sequential communications 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

Table 1. A Partial Summary of the OWL-S constructs

OWL-S Constructs	**Description**
process:Sequence	Executes a list of processes in order.
process:Split	Executes a bag of processes concurrently.
process:Split+Join	Executes a bag of processes concurrently with barrier synchronization.
process:Any-Order	Execute a bag of processes in any order but not concurrently.
process:Choice	Chooses between alternatives and executes.
process:If-Then-Else	Tests the *if-condition*. If *true* executes the "Then" branch, if *false* executes the "Else" branch.
process: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).

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 term 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 is set to be hot so that completion of all components are guaranteed.

- *Unordered*: All components must be executed. 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 DRS[1] or SWRL [12] or perhaps KIF[2]) properly to logical expression in LSCs.
- *Repeat-While and Repeat-Until*: Whether the test occurs at a fixed place within the iteration or runs asynchronously varies from subclasses to subclass of these classes. 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. 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*: 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 do the 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 such user-interface built with Visual Basic, and plays-out the corresponding LSCs according the user interaction through the interface.

4 Case Study

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

4.1 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.

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

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 book will be made.

Figure 2 is a RDF graph of the service 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.cs.man.ac.uk/~hwang/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 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>
```

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

[4] These variables are represented as `budget_ticket_Cost`, `budget_hotel_Cost` and `budget_total_Cost` in the ontology, respectively.

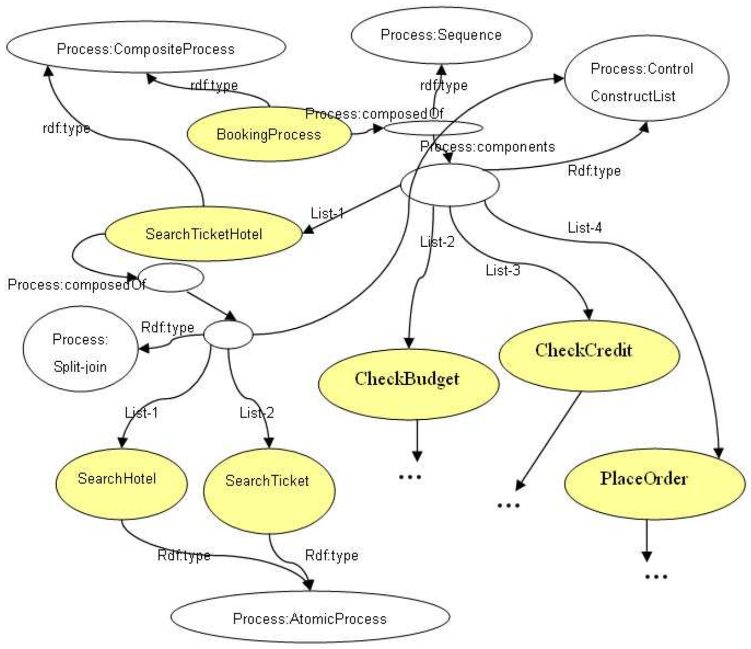

Fig. 2. Holiday booking System

```
      <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>
```

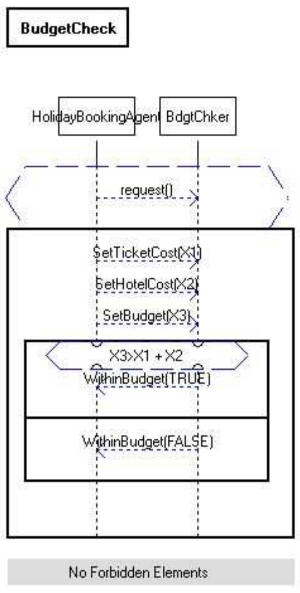

Fig. 3. LSC Example: Budget checking

Figure 3 shows an LSC universal chart capturing the necessary interactions between a service-using agent and a budget-checking agent cooperating in the above 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.

4.2 Simulation

Figure 4 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 will search for valid air tickets and hotels. Finally if such flights and hotel accommodation are available, it will proceed to book the flight and room.

Our simulation begins with building a simple Graphical User Interface (GUI) for 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 4. Play-Engine allows user-defined variables and external function through ActiveX DLLs. For

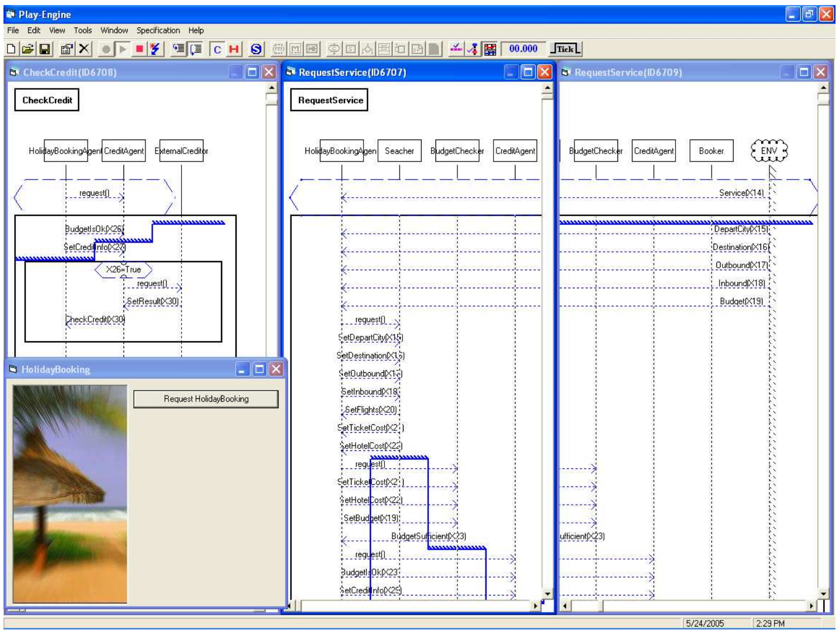

Fig. 4. Simulation Screen Shot

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 are popped 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 4, 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) are accessed exclusively.

5 Conclusion

In this paper, 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 [14]. 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 [9] can also be considered to represent OWL-S ontologies and their tool support, such as the FDR [16] or SPIN [10] model checkers, may also be used to perform verification tasks.

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.

Acknowledgement

The second author would like to thank Singapore Millennium Foundation[6] for the financial support. This work was supported in part by HyOntUse Project

[5] cf. `http://owlseditor.semwebcentral.org/index.shtml`
[6] cf. `http://www.smf-scholar.org/`

(GR/S44686) funded by the UK Engineering and Physical Science Research Council.

References

1. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):35–43, 2001.
2. D. Booth, M. Champion, C. Ferris, F. McCabe, E. Newcomer, and D. Orchard. Web Services Architecture. `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`, Feb. 2004.
3. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (WSDL) 1.1*. W3C, 1.1 edition, March 2001. `http://www.w3c.org/TR/wsdl`.
4. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, page 451. Kluwer, B.V., 1999.
5. V. Haarslev and R. Möller. *RACER User's Guide and Reference Manual: Version 1.7.6*, Dec. 2002.
6. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out. In *OOPSLA Companion*, pages 68–69, 2003.
7. D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. Technical Report MCS01-15, The Weizmann Institute of Science Rehovot, Israel, 2002.
8. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
9. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
10. G. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
11. I. Horrocks. The FaCT system. *Tableaux'98, LNCS*, 1397:307–312, 1998.
12. I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. `http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/`, May 2004.
13. ITU. *Message Sequence Chart(MSC)*, Nov 1999. Series Z: Languages and general software aspects for telecommunication systems.
14. H. Knublauch, R. W. Fergerson, N. F. Noy, and M. A. Musen. The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In *Proceedings of the Third International Semantic Web Conference (ISWC 2004)*,, Hiroshima, Japan, Nov. 2004.
15. M. K. Smith and C. Welty and D. L. McGuinness (editors). OWL Web Ontology Language Guide. `http://www.w3.org/TR/2004/REC-owl-guide-20040210/`, 2004.
16. A. W. Roscoe. *Theory and Practice of Concurrency*. International Series in Computer Science. Prentice-Hall, 1997.
17. The OWL Services Coalition. OWL-S: Semantic Markup for Web Services. `http://www.daml.org/services/owl-s/`, 2004.
18. UDDI. *Universal Description, Discovery, and Integration of Business for the Web*, October 2001. `http://www.uddi.org`.
19. W3C. *Simple Object Access Protocol (SOAP) 1.1*, 2000. `http://www.w3c.org/TR/SOAP`.