Monitoring execution of OWL-S web services*

Roman Vaculín, Katia Sycara
{rvaculin, katia}@cs.cmu.edu

The Robotics Institute, Carnegie Mellon University

Abstract. In this paper we describe mechanisms for execution monitoring of OWL-S web services. The described mechanisms are implemented as extensions of the OWL-S Virtual Machine which is a component that controls interactions between a client and web services. All extensions are driven by practical needs that arose as part of two projects involving OWL-S web services. Specifically, an event-based model for monitoring and logging is described that allows a human or software agent to analyze, replay or debug the execution. Next, we describe mechanisms for error handling and reporting which is not addressed by the OWL-S specifications. Finally, we describe the virtual machine introspection extensions motivated by needs of advanced software agents as brokers or process mediators that need to interact with web services dynamically.

1 Introduction

The main goal of Web Services is to enable and facilitate smooth interoperation of diverse software components in dynamically changing environments. Emerging Semantic Web Services standards as OWL-S [1] and WSMO [2] enrich web service standards like WSDL [3] and BPEL4WS [4] with rich semantic annotations to further facilitate flexible dynamic web services discovery, invocation and composition. Since all these tasks are expected to be performed fully or semi automatically by software agents, many practical problems arise. A software agent must be able to not only understand the semantics of the web service but it must also be able to interpret the course and the results of the execution and to deal with erroneous states. It must be able to understand and interpret the sources of problems so that it can recover or avoid the situation next time if possible. This poses challenges for both the semantic web services specification frameworks and for invocation tools. Execution monitoring mechanisms are needed to provide agents with appropriate information about the execution course and results.

* This research was supported in part by Darpa contract FA865006C7606 and in part by funding from France Telecom.

OWL-S represents one of main efforts in the semantic web services domain. OWL-S covers three areas: the Service Profile describes what the service does in terms of its capabilities and is used for discovery purposes; the Process Model specifies ways through which clients can interact with the service; the Grounding links the process model to the specific execution infrastructure (e.g., maps processes to WSDL operations and allows for sending messages in SOAP [5]). The elementary unit of the Process Model is an atomic process, which represents one indivisible operation that the client can perform by sending a particular message to the service and receiving a corresponding response. Processes are specified by means of their inputs, outputs, preconditions, and effects (IOPEs). Types of inputs and outputs are usually defined as concepts in some ontology or as simple XSD data-types. Processes can be combined into composite processes by using the various control constructs such as sequence, any-order, choice, if-then-else, etc. Besides control-flow, the process model also specifies a data-flow between processes. In the further text we assume that the reader is familiar with OWL-S.

Algorithms and software tools using OWL-S ontologies for discovery [6], invocation [7] and composition [8,9,10,11] were developed and are available. However, OWL-S does not provide explicit support for monitoring and errors handling.[1] Specific applications and tools are supposed to cover these areas. We will describe monitoring mechanisms that we implemented as extensions of the OWL-S Virtual Machine [7] which is a component that controls interactions between the client and web services. Specifically, the OWL-S Virtual Machine (OVM) executes the process model of a given service by going through the process model while respecting the OWL-S operational semantics [12] and invoking individual services represented by atomic processes. During the execution, the OVM processes inputs provided by the requester and outputs returned by the provider's services, realizes the control and data flow of the composite process model, and uses the grounding to invoke WSDL based web services when needed. The OVM is a generic execution engine which can be used to develop applications that need to interact with OWL-S web services.

[1] There is a kind of support for dealing with errors in terms of conditional results that can be used for representing erroneous outcomes, but we believe that this is only a very basic support. We will analyze this in detail in Section 4.

The main contribution of this paper is the description of an event-based monitoring extension to the OVM combined with introspection functionalities and errors handling. These extensions allow to perform different monitoring tasks such as logging, performance measuring, execution progress tracking, execution debugging or evaluations of security parameters. We provide details on monitoring and logging in Section 3. In Section 4 we define an approach to error handling that is used as part of the monitoring extension and point out a possible extension of OWL-S that would allow uniform error handling of different types of errors that can occur during the execution. Section 5 addresses the support for introspection in the OVM. In the last section, we conclude.

2 Problem domains

The need for monitoring, logging, error handling and execution introspection arose as a natural requirement in the context of two different projects where the OVM is used as an invocation engine of OWL-S web services.

In the POIROT project, machine learning techniques are used in order to learn and perform complex web service workflows, given a single demonstration example. The human expert is using web services to solve some complex problem, e.g., evacuation of wounded patients from the battlefield to a hospital. The solution of the given problem is recorded as a sequence of calls to individual web services that perform tasks such as looking up airports by geographic location, finding available flights to and from those airports, reserving seats on flights and reserving hospital beds at the destinations. Various components of the POIROT system use recorded observations in order to learn hierarchical task models and generalizations of these workflow traces by inferring task order dependencies, user goals, and the decision criteria for selecting or prioritizing subtasks and service parameters. The POIROT can dynamically access the same variety of semantically interoperable domain services as the user, which allows it to perform experiments to verify or falsify generated hypotheses by simulating the real services execution. The OVM is used as an execution component and is also part of the experiment execution module. As such, it must provide rich enough feedback to learning components to allow them to acquire new knowledge based on experimental results.

In the second project, OVM is used as part of the mediation / brokering component whose goal is to automatically reconcile discrepancies and incompatibilities between service requester and service provider assuming that the provider is (generally) able to satisfy the requester's needs. Since different types of mismatches between provider's and requester's process models are possible, the mediator component must be able to analyze both process models and to dynamically translate requester's messages and execute provider's process model to allow smooth interoperation. OVM is used to introspect and to dynamically execute the provider's process model.

Although the two projects are very different, from the perspective of services invocation and monitoring the requirements are very similar and complementary. The problems that we needed to address can be formulated as follows:

1. *Record the service calls and their outcomes during the execution of the process model(s).*
   The recorded sequence can be used as an observation for the POIROT learning components but it can be useful also for different purposes as, e.g., debugging of the process model. Depending on the specific purpose a different level of detail may be needed.
2. *Verify if a given sequence of calls (atomic process) can be "generated" by the composite process and/or replay a given sequence of calls if it is possible. If the execution fails, identify reasons for it.*
   The POIROT experimental component may be interested in replaying exactly the same sequence as the one performed by the human expert, or it may modify the sequence to verify some hypothesis. Similarly, in the process mediation scenario, we are interested in testing, if a given sequence of requester's calls can be satisfied by the provider's process model. In these cases the OVM must be able to record enough information to allow replaying of some execution and to identify reasons for possible execution failures.
3. *In the mediator / broker / client driven execution allow the client to see what steps (calls) can be chosen in each state of execution and/or allow the client to "simulate" or execute the chosen call. If the call cannot be executed or if it fails, provide an appropriate explanation.*
   The process model may be complex and may contain nondeterministic choices (*any-order* and *choice* control constructs) and the decision

of what choice is appropriate often depends on the execution context and on the application logic. Software agents enacting the client role of some process model must be able to decide, what choice to take, if more options are available. Since the OVM "knows" the execution context (e.g., values of variables, precondition evaluations, etc.), it can simplify the client's decision by providing introspection functionalities allowing the client to see what choices are available in the given execution context and filtering out those that are not available, e.g., due to unsatisfied preconditions.

3 Event based monitoring and logging

To solve the problem of monitoring during the process model execution, at least two questions must be answered: what exactly should be monitored and what (implementation) model should be chosen. While OWL-S itself does not address these issues, the clear semantics of the process model helps in answering the first question. By analyzing the process model and the grounding, it is possible to identify important events that might be monitored. The following list summarizes event types that occur during the execution of the process model:

- **Process call**: Presents probably the most important event type. An atomic process call can be recorded as one event together with its inputs and outputs values and effects. A simple and a composite process represent decomposition of a process into subprocesses. Therefore we represent them as two separate associated events: the start of the process and the end of the process. The start event is associated with input values and the end event with output values and effects.
- **Inputs assignment**: Input values of processes can be provided either by the user (client) of the process model or by the data binding that is used, e.g., an input or an output of some previous or ancestor process as the value source. We distinguish these two different situations as separate event types.
- **Outputs processing**: Outputs of atomic processes are obtained as a result of the service execution which is covered by the *process call* event type and so no new event type needs to be introduced. For simple and composite processes a new event type is needed to represent the fact that the output value of the process is obtained from some *output*

data binding (such as, in the case of the *OutputBinding* produced by the *Produce* pseudo-step).

- **Preconditions evaluation**: Represents preconditions evaluation of the process with variables values assigned and with the true or false status.
- **(Conditional) result evaluation**: Represents an evaluation of a result comprising the grounded *inCondition* expression, produced effects and output bindings. A special event type represents a situation when no result can be applied because the *inCondition* expression fails for all conditional results.
- **Control construct execution**: For each control construct one event type represents its start and one its end. Furthermore, we define specific event types for particular control constructs representing specifics of their semantics. For control constructs involving nondeterministic choices (*any-order* and *choice*) we define an event representing that a particular branch was chosen. For control constructs whose execution depends on an expression evaluation (*if-then-else*, *repeat-while*, *repeat-until*) the information representing this expression evaluation and the branch chosen is included in the starting event type. Further, we introduce event types that capture events as start and end of the iteration in loop control constructs and start and end of the branch in the *split* and *split-join* constructs.
- **Grounding events**: There can be different groundings for a given process model. Since currently only WSDL grounding is defined by OWL-S specifications, we identify only event types specific to WSDL grounding. The WSDL grounding defines mappings of atomic processes to WSDL operations and of inputs and outputs to WSDL messages and message parts. We define a separate event type for each type of the mapping, e.g., an event type for mapping of an input, an event type for mapping to a input message, etc.
- **Failures and erroneous events**: For different categories of errors specific event types are defined. We analyze error types and errors handling in Section 4.

Based on this analysis we defined a hierarchy of event types showed in Figure 1. Particular event types mentioned in the previous text are in the leaves of the taxonomy. For space reasons, the figure does not show all event types.

Fig. 1. Event types taxonomy. Only selected classes are displayed, particularly, specific *ExceptionEvent* types are not shown (see Section 4) and only some *ControlConstructEvent* types are shown.

It is important to note that described event types *are application independent* in two senses. First, they are derived only from the logic of the process model and therefore can be used in any application. Second, they are neutral to the purpose for which they can be used. So, for example, it is easy to imagine, that if the OVM emitted described events during the process model execution, they could be used for generating logs. If a different monitoring task needs to be performed as, e.g., performance analysis, the events could be used as well. Thanks to the application independence, the described events can serve as a sound basis for the mon-

itoring system. We adopted the event-based model [13] as the basis of the monitoring extension to the OVM.

3.1 Events handling

During the execution of the process model, OVM emits instances of described event types (*events*). Events can be processed by *event-handlers*. There can be one, several or none event handler for a given event type. The hierarchical organization of event types allows to define event-handlers with varying granularity, e.g., one event-handler can be defined for all instances of the *ProcessCallEvent* type (similarly to exception handling in object-oriented programming languages). The implementation of the OVM itself does not include any event-handlers and all events are ignored by default. Event-handlers are application specific and can be defined by the application programmer. Advantages of this model are its efficiency and flexibility. An application can define event-handlers only for event types that are important to it while ignoring the other.

Internally, events are represented as instances of OWL classes. This choice is quite natural, since details of an event can be specified by referring to relevant parts or including parts of the executed process model which is mainly defined in terms of OWL classes and instances. This is also convenient for OWL-S aware applications since they can easily interpret the content of events. So, for example, the event representing a call of some atomic process refers to the instance of this process in the process model and uses its inputs and outputs definitions when specifying their values. We defined an ontology[2] of event types with each particular event type represented by one OWL class. The ontology exactly corresponds to the hierarchy of event types introduced in Figure 1.

Figure 2 shows an instance of one event type emitted by the OVM during the execution. This event represents a call of the *isbnLookup* web service which searches for an ISBN given a book title as an input. The *isbnLookupPM* namespace refers to the process model of the *isbnLookup* web service and the *books* namespace refers to the books ontology that defines concepts such as *ISBN*. In this example, the event refers to the execution of the *&isbnLookupPM;isbnLookup* atomic process with "Crime and Punishment" as the input value of the *&isbnLookupPM;bookTitle* input parameter. The service returned an instance of the *books:ISBN*

[2] The ontology is available at `http://www.daml.ri.cmu.edu/owls/events.owl`

```
<AtomicProcessCallEvent>
  <timestamp>2007-03-12T12:35:12</timestamp>
  <process rdf:resource="&isbnLookupPM;isbnLookup"/>
  <input>
    <ParameterValueBinding>
      <toParameter rdf:resource="&isbnLookupPM;bookTitle"/>
      <dataValue>Crime and Punishment</dataValue>
    </ParameterValueBinding>
  </input>
  <output>
    <ParameterValueBinding>
      <toParameter rdf:resource="&isbnLookupPM;isbn"/>
      <objectValue>
        <books:ISBN>
          <books:value>978-0140621808</books:value>
        </books:ISBN>
      </objectValue>
    </ParameterValueBinding>
 </output>
</AtomicProcessCallEvent>
```

Fig. 2. An event instance: atomic process call event representing *isbnLookup* service call

class representing the 978-0140621808 ISBN as the value of the *&isbnLookupPM;bookTitle* output parameter.

3.2 Logging

With the event based monitoring and event-handlers we get the logging infrastructure almost for free. The log record of an execution session is stored as a sequence of OWL instances representing events generated during the execution. We developed a set of event handlers for the logging purposes which we use in our projects and which can be used as a general logging tool for OWL-S based application development. Depending on the particular purpose only some event-handlers are activated. So, for example, in the POIROT project, services are currently represented as atomic processes. Learning components therefore need to see only instances of the *AtomicProcessCallEvent* and possible *FailureEvents* in the log. In a different context, if we need to guarantee that the execution of a composite process model can be replayed by the OVM, additionally all instances of the *ControlConstructEvent* and *ProcessCallEvent* are recorded in the log. Finally, for the debugging purposes all generated events are being logged.

4 Dealing with errors

OWL-S does not define a model for error handling and reporting. Application level errors and failures as, for example, the situation when no ISBN is found for a given book title, are supposed to be handled by using conditional results. Other types of problems, for example, invocation errors, must be taken care of by an execution engine. This lack of support for explicit errors handling causes several problems:

- every application must define its own specific mechanisms for errors handling which leads to decreased interoperability
- an OWL-S execution engine is not able to distinguish erroneous states caused by application level errors from the normal flow which complicates, e.g., monitoring and execution evaluation
- OWL-S process model does not capture explicitly the WSDL error handling based on fault messages of the WSDL operations. This enforces specific application level solutions.

We present a solution that we incorporated into our event-based model. It solves the mentioned problems only partially. Solving the error handling in OWL-S in general would require a deeper analysis which is out of the scope of this paper.

4.1 Errors as part of the event-model

Similarly to event types that we identified in the previous section, also different erroneous situations can be identified:

1. *OWL-S processing errors*: Capture parsing / syntax level problems and problems with malformed OWL-S files of a given service.
2. *Service invocation errors*: For example, communication failure, serialization / deserialization error, no response, malformed response, response time-out, etc.
3. *Process level execution errors*: Include all erroneous situations that may occur during the execution of the process model and are caused by the discrepancies or inconsistencies on the process model level. For example, a required input is not provided by the client, a wrong input type is provided, the precondition of a process fails so that it cannot be executed, etc.
4. *Application level errors*: Erroneous states specific to the application logic of a web service as, e.g., no ISBN is found for a given book title.

These problems are solved by specifyig different results in the process model.

The first three categories of errors are application independent which allows us to define specific event types in our event types hierarchy representing particular erroneous situations. Figure 3 shows a snippet of the events taxonomy with event types representing exceptional states. Exception events instances contain context information specifying reasons for the error.

Figure 4 displays an example of a process level error, namely the exception event caused by providing a wrong input type of the *&isbnLookup;isbnLookup* atomic process. An instance of the *&books;Title* class is expected as the value of the *&isbnLookup;bookTitle* input parameter according to the process model definition, but the client provided the *&xsd;string* instead.

The main problem of application level errors is that there is no way for an invocation engine to identify an application level error because it is represented as a conditional result and it is not distinguished from normal results. Therefore, for example, it is not possible to emit appropriate exception events or to generate a log record that would declare an erroneous state. We believe that OWL-S should provide some support for explicitly distinguishing application level erroneous states (results) from normal ones. One way of doing this would be to use a different OWL class for representing a normal result and a different one for representing an erroneous result. Currently every result is represented as an instance of the *Result* class. If, let us say, the *FaultResult* were introduced in the

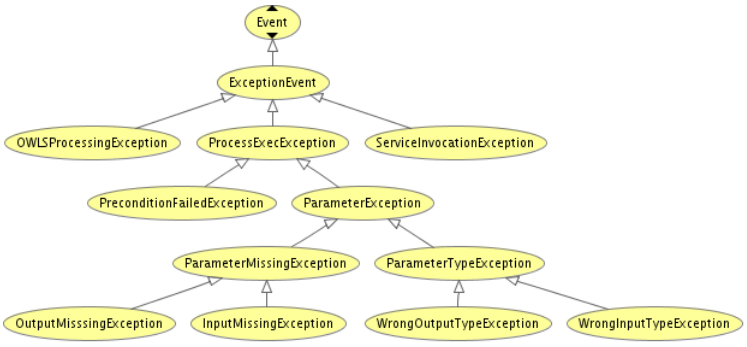

Fig. 3. Exception events taxonomy

```
<WrongInputTypeException>
  <timestamp>2007-03-12T12:47:23</timestamp>
  <process rdf:resource="&isbnLookup;isbnLookup"/>
  <parameter rdf:resource="&isbnLookup;bookTitle"/>
  <dataValue>Crime and Punishment</dataValue>
  <expectedType>&books;Title</expectedType>
  <invocationType>&xsd;string</invocationType>
</WrongInputTypeException>
```

Fig. 4. Example of an exception event

process model, an application designer could clearly separate normal and erroneous results. It would be also possible to define specific application exception event types hierarchies as subclasses of the *FaultResult*. This would allow an invocation component to handle application errors transparently in the same fashion as other types of exception events.

4.2 Runtime exceptions handling

When some erroneous situation occurs during the execution the OVM generates an instance of an appropriate exception event type. If an event-handler is defined for a given exception type, it is called. The event-handler can for example create an appropriate log record or it can inform a monitoring component. Besides, the normal execution flow is interrupted by throwing an ordinary exception of the programming language. Since the OVM is programmed in Java, we use the Java built-in exception handling mechanism. We did not address explicit exception handling in the OWL-S process model. In the OVM implementation, a thrown Java exception is propagated to the caller of the OVM. This behavior is appropriate in our context. However, it might make a sense to solve exception handling and/or compensation on the level of the process model (as for example BPEL4WS does). This would, however, require extension of OWL-S specifications as, for example, introducing the notion of exceptions in the process model and mechanisms of propagating of exceptions in composite processes.

5 Virtual machine introspection

Introspection functionalities are the last extension of the OWL-S virtual machine supporting monitoring tasks. By *introspection* we mean the ability of the OVM to provide the caller or the active event-handler with

information about the state of the execution during the runtime. In particular, the OVM allows examination of the current execution context, i.e., the values of inputs, outputs, local variables, state of precondition evaluation and the execution stack. Basically, the value of every variable and every expression with a specified URI can be inspected. This can be useful for example for debugging or tracing of the process model.

The OVM also provides information about what are the possible choices in terms of available next process calls in a given execution state. A composite process may include branching and choices, and in a given state, more than one choice may be possible. For example, a client of a fictive book selling service may at some point either search for a book, see the content of the shopping cart or proceed to checkout. Since the OVM is executing the process model and knows the execution context it can relatively easily evaluate what next process calls are possible. This information is particularly useful for components as service brokers or process mediators. If there is more than one choice available in a given state, the client can specify which one should be taken by the OVM.

Currently, the OVM supports only passive examination of the execution state. It is not allowed to modify the flow of execution by, for example, changing values of parameters.

6 Conclusions

In this paper we described an event-based monitoring model for OWL-S semantic web services. The main advantages of the model are its application independence, flexibility and extendibility. The model is easy to comprehend yet complex monitoring tasks can be performed by using it. It imposes only minimal constraints on the application and monitoring tools developers. Since it is tightly coupled with the OWL-S definitions of the process model, it allows to monitor virtually any aspect of the process model execution and to provide information in a way that is understandable by OWL-S aware clients. We applied the monitoring model to develop a logging facilities.

As part of the even-based monitoring problem we had to deal with error handling which is not covered by OWL-S specifications. Specifically, we figured out that it is impossible to identify application level errors in an application independent way because OWL-S does not support explicit specification of erroneous results/states. We suggested a relatively

simple extension in the form of introducing a new category of results representing erroneous states. This extension would allow applications to clearly distinguish normal results from errors and it would allow the invocation and monitoring tools to deal with application level errors in the same way as with any other errors that may occur during the execution. We are aware that the proposed extension covers only one part of the errors handling and processing which deserves a more comprehensive examination that would address such issues as exceptions handling in the process model or errors compensation.

References

1. The OWL Services Coalition: (Semantic Markup for Web Services (OWL-S)) http://www.daml.org/services/owl-s/1.1/.
2. Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web service modeling ontology. Applied Ontology **1(1)** (2005) 77 – 106
3. Christensen, E., Curbera, F., Weerawarana, G.M.S.: Web services description language (2001)
4. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., et al.: Business Process Execution Language for Web Services, Version 1.1. Specification, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems (2003)
5. SOAP: Simple object access protocol (SOAP 1.1). (http://www.w3.org/TR/SOAP)
6. Paolucci, M., Kawmura, T., Payne, T., Sycara, K.: Semantic matching of web services capabilities. (2002)
7. Paolucci, M., Ankolekar, A., Srinivasan, N., Sycara, K.P.: The DAML-S virtual machine. In Fensel, D., Sycara, K.P., Mylopoulos, J., eds.: International Semantic Web Conference. Volume 2870 of Lecture Notes in Computer Science., Springer (2003) 290–305
8. McIlraith, S., San, T.C.: Adapting Golog for composition of semantic web services. In Fensel, D., Giunchiglia, F., McGuinness, D.L., Williams, M.A., eds.: KR2002: Principles of Knowledge Representation and Reasoning. Morgan Kaufmann, San Francisco, California (2002) 482–493
9. Sycara, K., Paolucci, M., Ankolekar, A., Srinivasan, N.: Automated discovery, interaction and composition of semantic web services. Journal of Web Semantics **1 (1)** (2004) 27–46
10. Sirin, E., Parsia, B.: Planning for semantic web services. In: Semantic web services workshop at 3rd international semantic web conference (iswc2004). (2004)
11. McDermott, D.: Estimated-regression planning for interactions with web services (2002)
12. Ankolekar, A., Huch, F., Sycara, K.P.: Concurrent semantics for the web services specification language DAML-S. In Arbab, F., Talcott, C.L., eds.: COORDINATION. Volume 2315 of Lecture Notes in Computer Science., Springer (2002) 14–21
13. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley (2002)