Semantic Web Services Monitoring: An OWL-S based Approach*

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 semantic web services, based on OWL-S. The use of semantic descriptions and ontologies is a valuable extension to current SOA conceptualizations. The described mechanisms are implemented as extensions of the OWL-S Virtual Machine that we have previously developed. The OWL-S Virtual Machine is a component that controls the interactions between a client and the web service it uses. The presented extensions are a result of practical requirements that arose in the course of involvement with two projects that utilize OWL-S based semantic web services. In particular, we present an event-based model for monitoring and logging interactions. The interaction trace allows a human or software agent to analyze, replay or debug the execution. Additionally, we describe proposed extensions to current specification in OWL-S for reporting and handling of errors. Finally, we describe extensions to the OWL-S Virtual Machine based on introspection to enable dynamic interactions with process mediators.

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 [24], WSMO [18] and SAWSDL [5] enrich web service standards like WSDL [4] and BPEL4WS [1] 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 understand the semantics of the web service and 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. Furthermore, since web services are often used as part of complex processes models and workflows, the need for analyzing, diagnosing, simulating and optimizing of such processes models arises.

This poses challenges for both the semantic web services specification frameworks and for invocation tools. Execution monitoring mechanisms are needed to provide human or software agents with appropriate information about the execution course and results. Information provided by monitoring mechanisms can be used either during the execution to support a dynamic response to the given execution course, or after the execution is finished for purposes of analysis and auditing. In both cases different levels of abstraction and details may be needed depending on the given problem. On the business level, typically highly summarized information related to the process execution is of interest, as for example, the profitability of individual services, the average usage of services, or the overall workflow effectiveness. On the technical level, e.g., the performance analysis or debugging tasks require much more detailed information about the execution process. The monitoring can be also useful to support measuring and evaluation of Quality of Services (QoS) metrics that are required by Service Level Agreements (SLA). To be able to deal with these possibly very diverse information needs, we propose flexible event-based monitoring mechanisms that produce a semantic interaction trace consisting of relevant execution related information. Such a semantic interaction trace allows software agents to respond to emerging situations during the execution process. Additionally, complex queries can be used to retrieve aggregate information from the interaction trace that might be needed for analytical purposes.

In this paper, we focus on the monitoring mechanisms of semantic web services based on OWL-S. OWL-S represents one of main efforts in the semantic web services domain. Algorithms and software tools using OWL-S ontologies for discovery [16], invocation [15] and composition [10, 23, 20, 9] were developed and are available. However, the current specification OWL-S does not provide explicit support for monitoring and errors handling. Specific appli-

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

cations and tools are supposed to cover these areas. We will describe monitoring mechanisms that we implemented as extensions of the OWL-S Virtual Machine (OVM) [15] which is a component that controls interactions between the client and web services.

The main contribution of this paper is the description of event-based execution monitoring and errors handling mechanisms for semantic web services based on OWL-S. We also describe extensions to the OVM implementing described monitoring mechanisms in combination with introspection functionalities. These extensions allow to perform different monitoring tasks such as logging, performance measuring, execution progress tracking, execution debugging or evaluations of security and other QoS parameters.

The rest of the paper is organized as follows. In Section 2 we briefly introduce the OWL-S and the OWL-S Virtual Machine. In Section 3 we describe practical requirements for monitoring and errors handling that arose in the course of involvement with two projects that utilize OWL-S based semantic web services. We provide details on monitoring and logging in Section 4. In Section 5 we define an approach to error handling that is used as part of the monitoring extension and point out a possible extension to current OWL-S specification that would allow uniform error handling of different types of errors that can occur during the execution. Section 6 addresses the support for introspection in the OVM. In the last section, we conclude.

2 Overview of OWL-S and the OWL-S Virtual Machine

OWL-S [24] is a Semantic Web Services description language, expressed in OWL [3]. OWL-S covers three areas: web services capability-based search and discovery, specification of service requester and service provider interactions, and service execution. The Service Profile describes what the service does in terms of its capabilities and it is used for discovering suitable providers, and selecting among them. The Process Model specifies how clients can interact with the service by defining the requester-provider interaction protocol. The Grounding links the Process Model to the specific execution infrastructure (e.g., maps processes to WSDL [4] operations and allows for sending messages in SOAP [22]). Corresponding Profiles, Process Model and Groundings are connected together by an instance of the Service class that is supposed to represent the whole service.

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

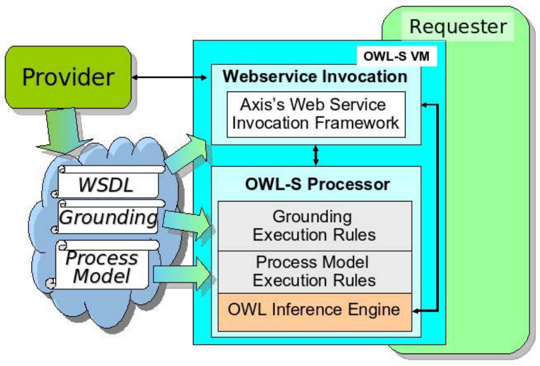

Figure 1. The OWL-S Virtual Machine Architecture

usually defined as concepts in some ontology or as simple XSD data-types. The process preconditions must all hold in order for the process to be successfully invoked. After the process is invoked, the outputs are produced and its effects are applied to change the state of the world. OWL-S introduces the term *result* to refer to coupled outputs and effects. The actual result (i.e. outputs and effects) can depend on conditions that hold true in the actual world state at the time the process is performed. For example, a selling service may require as a precondition a valid credit card and as an input the credit card number and the expiration date. As an output it generates a receipt, and as an effect the card is charged, in case that the credit card limit is not exceeded. Otherwise, a failure message is generated as an output and no effect is applied to the world. 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.

A tool for execution of OWL-S web services must be able to interpret the Process Model of the service according to its semantics and provide a generic mechanism for invocation of web services represented as atomic processes in the Process Model. The OWL-S Virtual Machine (OVM) [15] is a generic OWL-S processor that allows Web services and clients to interact on the basis of the OWL-S description of the Web service and OWL ontologies. 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 [2] 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.

The architecture of the OVM and its relation with the rest of the Web service is presented in Figure 1. On the left side the provider is displayed together with its OWL-S Process Model, Grounding and WSDL description that together define how clients can interact with this service. The OVM is displayed in the center of the picture. It is logically divided in two modules: the first one is the OWL-S Processor which uses the OWL-S Inference Engine and a set of rules implementing the operational semantics of the OWL-S Process Model and Grounding to manage the interaction with the provider. The second component is the Web service Invocation module that is responsible for the information transfer with the provider. Finally, the OVM is shown as a part of the requester which can use it to interact with the provider.

3 Motivating applications

Monitoring is recognized as a natural part of the Service-oriented architectures. In the SOA systems the main stress is usually put on the monitoring performance and availability metrics. The monitoring sub-system should be able to support tasks such as enforcement of the SLA, notifications and auditing of service performance, alert-based reporting on the level of adherence to the SLA, or sending automatic notifications and allow graceful exception handling when the SLAs break down. In the broader context, web services realizing Service-oriented architectures can be seen as a part of Business Process Management (BPM). BPM is approaching the management and the execution of IT-supported business operations from a business expert's view rather than from a technical perspective [21]. In the context of BPM and the Business Process Analysis (BPA) [25] monitoring is supposed to support mainly optimization, reengineering and fine-tuning of existing process models.

Except for these general tasks we also identified a set of specific needs for monitoring, logging, error handling and execution introspection that arose as natural requirements in the context of two different projects where the OVM is used as an invocation engine of OWL-S web services. In the following paragraphs we first briefly describe relevant aspects of the two projects and then we summarize resulting requirements.

In the POIROT project, computational modules use machine learning techniques in order to learn to perform complex web service workflows, given a single demonstration example. The human expert is using a graphical user interface to solve some complex problem, e.g., evacuation of wounded patients from the battlefield to a hospital. The problem can be solved by combining primitive operations available in the GUI. Primitive operations are realized as semantically annotated domain 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. The solution of the given problem is recorded as a sequence of calls (a trace) to individual web services. The computational learners in the POIROT system use these recorded traces 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 system 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, the OVM is used as part of the mediation / brokering component whose goal is to automatically reconcile discrepancies and incompatibilities between a service requester and a 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. The OVM is used in the mediation component to allow the dynamic execution of the the provider's process model.

Since neither the requester nor the provider are known in advance, the mediation component must be able to dynamically execute the provider's process model depending on incoming requester's messages and also be able to interpret the flow of the execution and its results. To allow this, monitoring and execution introspection support must be provided in the OVM.

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 (interaction trace) 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 or the post-execution analysis. Depending on the specific purpose a different level of detail may be needed.
2. *Replay or simulate a given sequence of calls if it is pos-*

sible.
The POIROT experimental component may be interested in replaying exactly the same sequence of calls as the one performed by the human expert, or it may modify the sequence to verify some hypothesis. In these cases the OVM must be able to record enough information to allow replaying of some execution. Also the OVM implementation must provide support for replaying a given interaction trace.

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.*
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.
4. *Verify if a given sequence of calls (atomic process) can be "generated" by the composite process.*
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. This can be nontrivial in cases of complex process models.
5. *If the process model or the service call cannot be executed or if it fails, provide an appropriate explanation and identify reasons for it if possible.*
For the the purposes of dynamic execution and post-execution analysis it is important to be able to identify execution failures and their reasons. Failures must be represented explicitly along with the context information as a part of the interaction trace. Additionally, during the dynamic invocation, e.g., in the mediation scenario, the execution introspection functionalities would allow a software agent to inspect the execution context in detail and to identify possible reasons for the failure.

4 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. For each process type specific event types are defined representing its start and end. Start events are associated with input values and end events additionally with produced output values and effects. A simple and a composite process represent decomposition of a process into subprocesses while an atomic process represents an execution of an existing web service operation.
- **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* (i.e., the dataflow of the process model specifies that the output is produced by some previous processes).
- **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[1], 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*

[1]The *inCondition* property specifies a condition under which a particular result is produced.

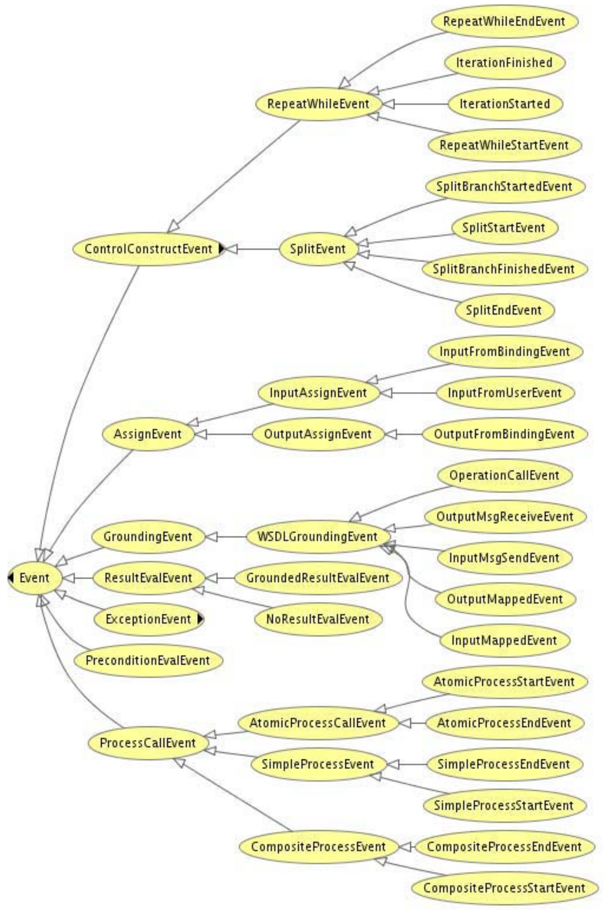

Figure 2. Event types taxonomy. Only selected classes are displayed, particularly, specific *ExceptionEvent* types are not shown (see Section 5) and only some *ControlConstructEvent* types are shown.

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.
- **Failures and erroneous events**: For different categories of errors specific event types are defined. We analyze error types and errors handling in Section 5.

Based on this analysis we defined a hierarchy of event types showed in Figure 2. 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.

It is important to note that described event types *are application independent* in two senses. First, event types are derived only from the logic of the process model and therefore can be used in any application. Second, event types 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 same events could be used as well. Thanks to the application independence, the described event types can serve as a sound basis for the monitoring system.

As an implementation model for event processing we adopted the event-based model [8, 11] which we implemented as an extension to the OVM. During the execution of the process model, the 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. For example, one event-handler can be defined for all instances of the *ProcessCallEvent* type. This event-handler will be invoked also when an instance of the *AtomicProcessCallEvent* is emitted, because the *AtomicProcessCallEvent* type is defined as a subclass of the *ProcessCallEvent* type. The described mechanism is similar 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 others.

4.1 Events ontology

We defined an ontology of event types[2] with each particular event type represented by one OWL class [3]. The ontology exactly corresponds to the hierarchy of event types introduced in Figure 2. Every emitted event is thus represented as an instance of the OWL class representing the event type of the emitted event. Such a choice is quite natural for several reasons.

First, details of an event can be specified by referring to or by including relevant parts of the executed process model which is mainly defined in terms of OWL classes and instances. So, for example, the event representing a call of some atomic process can refer to the instance of this process in the process model and use its inputs and outputs

[2]Available at `http://www.daml.ri.cmu.edu/owls/events.owl`

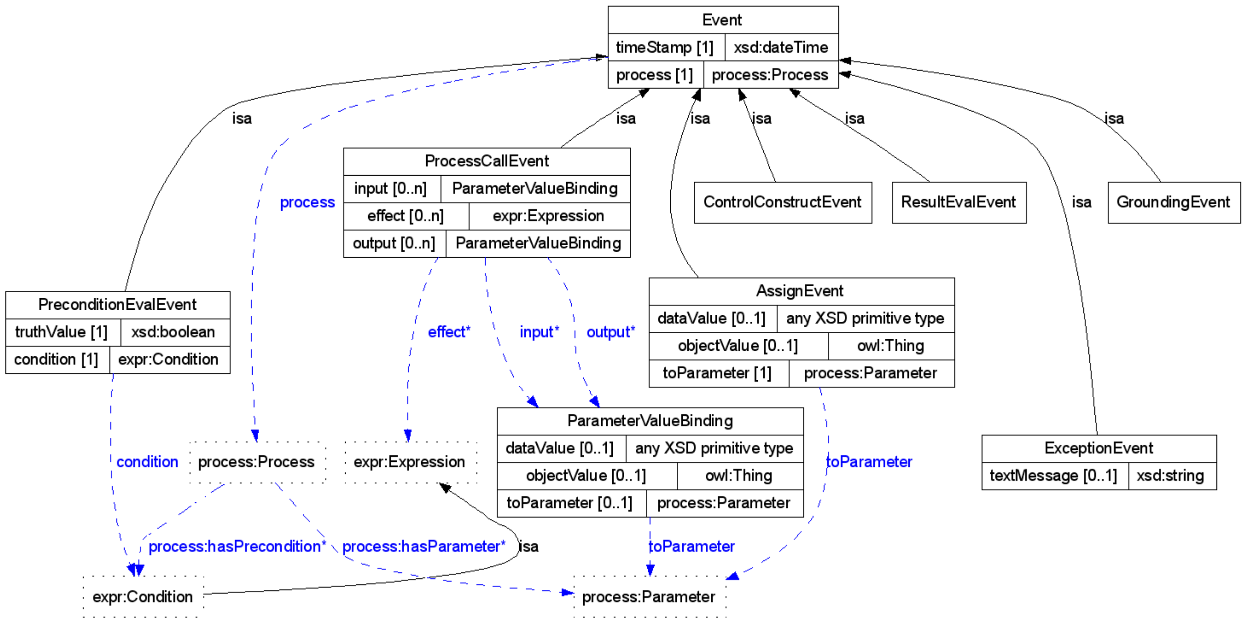
Event
timeStamp [1] | xsd:dateTime
process [1] | process:Process
isa
ProcessCallEvent
input [0..n] | ParameterValueBinding
effect [0..n] | expr:Expression
output [0..n] | ParameterValueBinding
ControlConstructEvent
ResultEvalEvent
GroundingEvent
AssignEvent
dataValue [0..1] | any XSD primitive type
objectValue [0..1] | owl:Thing
toParameter [1] | process:Parameter
PreconditionEvalEvent
truthValue [1] | xsd:boolean
condition [1] | expr:Condition
process
effect*
input*
output*
condition
toParameter
ParameterValueBinding
dataValue [0..1] | any XSD primitive type
objectValue [0..1] | owl:Thing
toParameter [0..1] | process:Parameter
ExceptionEvent
textMessage [0..1] | xsd:string
process:Process
expr:Expression
process:hasPrecondition*
process:hasParameter*
expr:Condition
process:Parameter

Figure 3. Direct subclasses of the Event class with their properties

definitions when specifying their values.

Second, OWL representation of events is convenient for OWL-S aware applications since at least a basic OWL machinery must be already available. This allows an easy interpretation of the events content without enforcing many implementation changes.

Third, all events along with their content are described in terms of explicitly defined ontologies which has many advantages. Due to a clearly defined semantics and standardised serializations events expressed as OWL instances can be easily processed and shared by software agents and other software tools. During the execution, it is possible to employ flexible event handlers that could for example benefit from using reasoning about processed events. After the execution is finished, an interaction trace containing the events emitted during the execution can be used for post-execution analysis. Compex filtering and querying techniques exploiting the rich semantic interaction traces can be used to analyze the process model and its execution.

Figure 3 presents a structure of event types defined in the events ontology. For space reasons only direct subclasses of the *Event* class are shown. Every event type is displayed as a solid box with the name in its heading and the list of its properties with cardinalities and range type specification. Solid arrows with the "*isa*" label represent subclassing relation while dashed arrows represent relations between classes. Classes defined in other ontologies are identified by an appropriate namespace and are shown as dotted boxes. For example, *process:Parameter* means that the *Parameter* class is defined in the OWL-S *process* ontology.

Each *Event* instance is associated with a *timeStamp* referring to the time when the event was emitted. Since an event is always emitted during the execution of some process, the *process* parameter is used to refer to such a process. When an event is emitted in a nested process, the most inner process is used as a value of the *process* parameter.

The *ProcessCallEvent* type defines properties for specifying *input* and *output* values and *effects* of the executed process. The *ParameterValueBinding* class used as range of the *input* and the *output* property represents a value assigned to an input or to an output parameter of the process. When a parameter is defined as a primitive XSD type, the *dataValue* property is used to refer to its value, otherwise the *objectValue* is used to refer to values that are instances of OWL classes. Figure 5 shows an example event with inputs and outputs assigned.

The *AssignEvent* type (representing details about assigning values either to inputs or outputs) uses the same logic for specifying parameter values as the *ParameterValueBinding* class.

The *PreconditionEvalEvent* type represents the precondition evaluation and refers to the precondition expression with values assigned (the *condition* property) and to the truth value (the *truthValue* property).

Finally, the *ExceptionEvent* defines a *textMessage* property containing a text message with detail information about the exception. For the remaining event types that are not

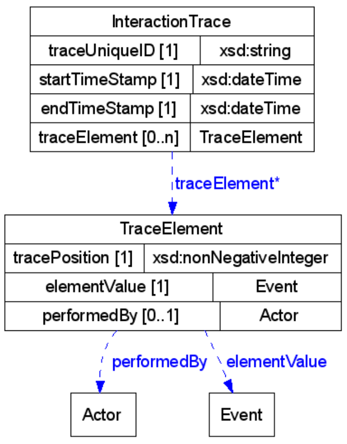

Figure 4. Event container classes

```
<AtomicProcessEndEvent>
  <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>
</AtomicProcessEndEvent>
```

Figure 5. An event instance: atomic process call end event representing successful *isbnLookup* service call

shown in Figure 3 properties are defined in a similar fashion.

In addition to event types, the events ontology also defines classes for storing interaction traces. Basically, an interaction trace is represented as a list of all events that were emitted during the process execution with some more information added. The *InteractionTrace* class is defined as an ordered list of *TraceElement* instances that associate emitted events with the *Actor* responsible for emitting the event (see Figure 4). Each *InteractionTrace* instance is identified by a *traceUniqueID* and has start and end time associated.

Example: Figure 5 shows an instance of one event emitted by the OVM during the execution of the *isbnLookup* service. 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* class representing the 978-0140621808 ISBN as the value of the *&isbnLookupPM;bookTitle* output parameter.

4.2 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 an instance of the *InteractionTrace* containing 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.

5 Dealing with errors

OWL-S does not define a model for error handling and reporting. Application level errors and failures such as 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, e.g., 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 above 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.

5.1 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 6 shows a snippet of the events taxonomy with event types representing exceptional states. Exception events instances contain context information specifying reasons for the error.

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 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.

```
<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>
```

Figure 7. Example of an exception event

Example: Figure 7 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.

5.2 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 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.

6 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,

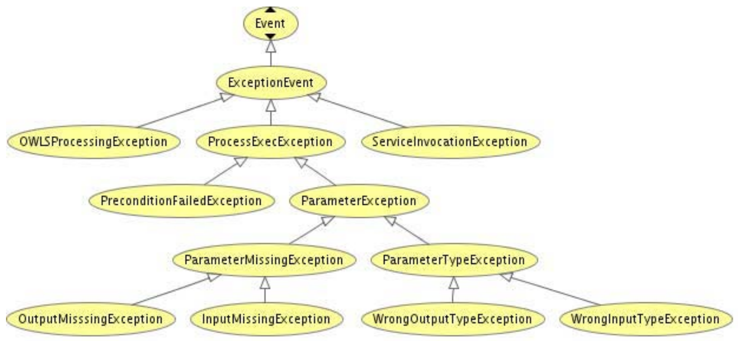

Figure 6. Exception events taxonomy

the value of every named variable and every named expression in the process model 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 fictitious 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.

7 Related work

Extensive work has been done in the area of events processing, passing and monitoring. A general coverage of events-based systems is provided in [8] and [11]. Distributed middleware systems based on CORBA, JMS and Web Services standards as WS-Eventing and WS-Notification typically include a monitoring subsystem and tools for analyzing logged events. As we mentioned earlier, such systems are typically concerned with monitoring of performance, availability and other SLA metrics. The Web Service Level Agreement (WSLA) framework [7] is targeted at defining and monitoring SLAs for Web Services. This framework defines mechanisms for specifying and monitoring Service Level Agreements. SLA monitoring issues in multi-provider environments are described in [14] and [13]. Sahai et. al. [19] developed an automated and distributed SLA monitoring engine that allows definition of SLAs and their automatic monitoring and enforcement. These existing systems and tools usually use different formats for reading/storing log files and present their results in different ways. This problem is addressed by the ProM framework (pluggable environment for process mining) [26]. The goal of the ProM is to define independent algorithms for process mining. ProM uses a generic format for representation and storing of events and allows to import logs from several existing commercial systems. The general problem of current systems is the lack of machine processable semantics as identified in [6]. The only work know to us that addresses the problem of semantic process monitoring is presented in [17]. An ontology for process monitoring and mining is used in the context of the Super project that builds on WSMO framework [18]. Monitoring issues are also addressed in works dealing with workflow [12] and process adaptation [27].

8 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] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, et al. Business Process Execution Language for Web Services, Version 1.1. 2003.

[2] A. Ankolekar, F. Huch, and K. P. Sycara. Concurrent semantics for the web services specification language DAML-S. In F. Arbab and C. L. Talcott, editors, *COORDINATION*, volume 2315 of *Lecture Notes in Computer Science*, pages 14–21. Springer, 2002.

[3] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Owl web ontology language reference, 10 February 2004. W3C Recommendation, http://www.w3.org/TR/owl-ref/.

[4] E. Christensen, F. Curbera, and G. M. S. Weerawarana. Web services description language, 2001.

[5] J. Farrell and H. Lausen. Semantic annotations for wsdl and xml schema, 2006. http://www.w3.org/2002/ws/sawsdl/spec/, working draft.

[6] M. Hepp, F. Leymann, J. Domingue, A. Wahler, and D. Fensel. Semantic business process management: A vision towards using semantic web services for business process management. In F. C. M. Lau, H. Lei, X. Meng, and M. Wang, editors, *ICEBE*, pages 535–540. IEEE Computer Society, 2005.

[7] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *J. Network Syst. Manage*, 11(1), 2003.

[8] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[9] D. McDermott. Estimated-regression planning for interactions with web services, 2002.

[10] S. McIlraith and T. C. San. Adapting Golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M.-A. Williams, editors, *KR2002: Principles of Knowledge Representation and Reasoning*, pages 482–493. Morgan Kaufmann, San Francisco, California, 2002.

[11] G. Muhl, L. Fiege, and P. R. Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag, New York, Aug. 2006.

[12] R. Müller, U. Greiner, and E. Rahm. Agent work: a workflow system supporting rule-based workflow adaptation. *Data Knowl. Eng.*, 51(2):223–256, 2004.

[13] C. Overton. On the Theory and Practice of Internet SLAs. *Journal of Computer Resource Measurement*, 106:32–45, 2002.

[14] C. Overton and E. Siegel. Experiences with Internet measurements and statistics, Computer Measurement Group. *Journal of Computer Resource Measurement*, 106:4–14, 2002.

[15] M. Paolucci, A. Ankolekar, N. Srinivasan, and K. P. Sycara. The DAML-S virtual machine. In D. Fensel, K. P. Sycara, and J. Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2003.

[16] M. Paolucci, T. Kawmura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *First Int. Semantic Web Conf.*, 2002.

[17] C. Pedrinaci and J. Domingue. Towards an ontology for process monitoring and mining. In *Workshop: Semantic Business Process and Product Lifecycle Management (SBPM 2007), 4th European Semantic Web Conference (ESWC 2007)*, 2007.

[18] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77 – 106, 2005.

[19] A. Sahai, V. Machiraju, M. Sayal, A. van Moorsel, and F. Casati. Automated SLA Monitoring for Web Services. *IEEE/IFIP DSOM*, 2002, 2002.

[20] E. Sirin and B. Parsia. Planning for semantic web services. In *Semantic web services workshop at 3rd international semantic web conference (iswc2004)*, 2004.

[21] H. Smith and P. Fingar. Business Process Management: The Third Wave. 2003.

[22] SOAP. Simple object access protocol (SOAP 1.1). *http://www.w3.org/TR/SOAP*.

[23] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated discovery, interaction and composition of semantic web services. *Journal of Web Semantics*, 1 (1):27–46, 2004.

[24] The OWL Services Coalition. *Semantic Markup for Web Services (OWL-S)*. http://www.daml.org/services/owl-s/1.1/.

[25] W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske. Business process management: A survey. In W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske, editors, *Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2003.

[26] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst. The proM framework: A new era in process mining tool support. In G. Ciardo and P. Darondeau, editors, *ICATPN*, volume 3536 of *Lecture Notes in Computer Science*, pages 444–454. Springer, 2005.

[27] K. Verma, P. Doshi, K. Gomadam, J. Miller, and A. Sheth. Optimal adaptation in web processes with coordination constraints. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 257–264, Washington, DC, USA, 2006. IEEE Computer Society.