Specifying and Monitoring Composite Events for Semantic Web Services*

Roman Vaculín, Katia Sycara
{rvaculin, katia}@cs.cmu.edu
The Robotics Institute, Carnegie Mellon University

Abstract

Execution monitoring of complex web service process models is critical for effective management and control of web services based systems. During the composite process model execution or as part of the after-execution analysis complex event patterns (called composite events) consisting of various primitive events need to be detected. In this paper we introduce the concept of primitive semantic events that are used for monitoring of semantic web services, based on OWL-S. Next, we describe mechanisms for specification and detection of composite events. We present a language based on an event algebra combined with semantic event-filtering expressions using description logics atoms enriched with OWL datatypes and SWRL built-ins. Such a language can be used for specification of composite events allowing a detection of complex event patterns in the flow of primitive events. Furthermore, the semantic filtering allows detection of such events that would otherwise be impossible without the use of semantic descriptions. We also discuss detection mechanisms suitable for runtime execution and after-execution analysis.

1 Introduction

Emerging Semantic Web Services standards as OWL-S [18], WSMO [16] and SAWSDL [5] enrich web service standards like WSDL and BPEL4WS with rich semantic annotations to facilitate flexible dynamic web services discovery, invocation and composition. While the advantages of semantically annotated web services were recognized in the context of service discovery and composition, little effort has been invested into studying possibilities of semantic web services monitoring (semantic monitoring). The importance of powerful monitoring techniques increases as operating environments of web services become more dynamic and web services based information systems are expected to work in an autonomous or semi-autonomous fashion.

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. For example, during the execution, the monitoring can be used to support measuring and evaluation of Quality of Services (QoS) metrics that are required by Service Level Agreements (SLA). 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. The later scenario finds its applications mainly in areas as (Semantic) Business Process Management [7] and Process Mining [14].

For many applications simple detection of individual events emitted by various components of the systems is a good enough solution. However, often complex events patterns (called composite events) need to be detected. In this paper, we focus on primitive and composite event specification and detection mechanisms suitable for monitoring of semantic web services. The main contribution of this paper is the description of a language for specification of composite event patterns based on the event algebra developed originally in the context of active databases [4]. To support detection of events with semantically rich content that are emitted during the execution of semantic web services process model, we augmented the event detection algebra with semantic filtering. Specifically, the filtering is based on expressions in the form of description logics atoms enriched with OWL [2] datatypes and SWRL [8] built-ins. We discuss detection mechanisms and suitability of the language for specific detection scenarios. We show that a restricted combination of detection algebra with semantic filtering is rich enough to allow detection of event patterns during the runtime with acceptable memory and time requirements. At the same time, an unrestricted combination is useful during the after-execution analysis since it allows to identify more

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

complex event patterns in recorded semantic traces possibly over many execution sessions. OWL-S based semantic web services and derived primitive events are used as an underlying platform. However, the described language does not depend on OWL-S and can be easily adapted for other semantic web services frameworks, such as WSMO.

The rest of the paper is organized as follows. Section 2 briefly introduces OWL-S and the OWL-S Virtual Machine that executes OWL-S web services. In Section 3 we describe benefits of semantic monitoring and introduce example problems that can be addressed by semantic monitoring and events detection. Section 4 defines primitive events specific for OWL-S web services. In Section 5 we introduce the event detection algebra. In Section 6 we show how to extend the event detection algebra with semantic event filtering. In Section 7 we discuss event detection techniques. In Section 8 we review related work and in Section 9 we conclude and point out directions for future work.

2 Overview of OWL-S concepts

OWL-S [18] is a Semantic Web Services description language, expressed in OWL [2]. 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. 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 operations and allows for sending messages in SOAP).

An 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. 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. 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.

The OWL-S Virtual Machine (OVM) [13] 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 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 OVM uses an event-based model [10] as the basis of monitoring implementation. During the execution of the process model, the OVM emits various events specific to the state of process model execution. Events can be processed by event-handlers. Each event handler is associated with an event pattern which specifies, when the handler is triggered. In this paper, we focus on specification of appropriate event patterns and on their detection.

3 Benefits of semantic monitoring

In most works on event monitoring and filtering, the monitored system and its components emit event instances during its lifetime. Emitted event instances (often called primitive events) are typically characterized by an event type. Primitive events are usually directly derived from the system implementation and are represented on the syntactic level. Thus, detection mechanisms are restricted only to event types detection and syntactic parameters matching or comparison. Furthermore, often no explicit declarative specification of event types and their parameters is available. Reasons for the lacking semantics in emitted events partly spring from the lack of declarative semantic descriptions of the monitored components itself.

On the contrary, semantic web services frameworks provide means for explicit specification of web service capabilities, interfaces and interaction protocols. This is typically done by annotating web services with semantic annotations using concepts with a clear semantics defined in ontologies. Semantic descriptions can be also used for describing event types and event instances emitted during interactions with semantic web services. Similar to traditional monitoring approaches, event types can be also organized in taxonomy. However, in the semantic monitoring approach event types taxonomy and event parameters are defined in an ontology. Additionally, also the data associated with an event instance can be annotated by ontology concepts. Such a choice has several advantages. First, due to a clearly defined semantics and standardized serializations of ontologies, events and their content can be easily processed and shared by software agents and various applications. Sec-

Operation	Input names & types	Output name & type
Login	username(xsd:string), password(xsd:string)	status(xsd:boolean)
Logout	username(xsd:string)	status(xsd:boolean)
LookupItem	category(ItemCategory), name(ItemName)	item(ItemDetails)
AddItemToCart	id(ItemID)	id(CartId)
BuyAndShipItems	id(CartID), country(Country), city(City), street(Street)	confirmation(CfID)

Table 1. Operations of a shopping service

ond, instead of pure syntactic matching during event detection semantic reasoning can be used to support more flexible event detection. Finally, after the execution is finished, an interaction trace containing the events emitted during the execution and complex filtering and querying techniques exploiting the rich semantic interaction trace can be used for post-execution analysis.

In order to demonstrate the benefits of semantic event detection, we introduce a simple semantically annotated web service realizing an electronic shop. We assume that the web service supports operations defined in Table 1 (represented as atomic processes in OWL-S). For simplicity, we describe only input parameters and a return parameter type of each operation. We assume that all parameter types (e.g., *ItemCategory*, *ItemID*, *Country*, etc.) refer to concepts defined in a domain ontology. A client of this service can communicate with it according to the following informal process model definition: a transaction must start with the *Login* operation and end with the *Logout* operation. In between, *LookupItem*, *AddItemToCart* and *BuyAndShipItems* can be called repeatedly. OWL-S definitions of these services are defined in the *shoppingService* namespace. Examples in the following sections will often refer to this web service.

In the following paragraph we illustrate types of event patterns that might be of interest during execution monitoring or as part of the after-execution analysis.

1. **Event patterns using primitive events only:**
   (a) Detect every call of a given operation (e.g., *Logout*).
   (b) Detect when a particular result is produced, e.g., *Login* fails since the username cannot be verified.
   (c) Filter service calls with a given parameter type, e.g., *LookupItem* calls with the *category* parameter that is an instance of *Book* class.
2. **Complex event patterns:**
   (a) Detect repeated occurrence of some event within a certain time, e.g., 3 unsuccessful *Login* calls within 2 minutes.
   (b) Detect situations when the customer logs out without buying anything.
   (c) Detect service calls taking longer than a specified time. As a result a QoS metric might be updated.
3. **Event patterns that can be useful during the off-line post-execution analysis:**
   (a) Identify US customers shopping for *Books* that spent more than $1000 within 3 days.
   (b) Analyze popularity of some workflow (specified by some pattern and its features).
   (c) Analyze efficiency of a workflow (e.g., time to buy) or it effectiveness (i.e., if a given sequence of calls leads to purchasing a product).

Some of the described event patterns cannot be easily detected without events and their parameters being represented as instances of concepts in an ontology. For example, patterns 1c and 3a rely on the fact that parameter types are organized in a taxonomy and that ontology reasoning mechanisms can be used to check if the parameter value is an instance of required parameter type.

4 Primitive semantic events

A *primitive event occurrence* is an instantaneous, atomic occurrence of an interest at a point in time [4]. Primitive event occurrences are directly emitted by the system or its components. Each primitive event occurrence is an instance of some *event type* and possibly can have additional information in the form of parameters associated with it. In the context of semantically annotated web services, it is beneficial to define primitive event types as concepts in an ontology and occurrences of primitive events as instances of ontology concepts. Every emitted event is thus represented as an instance of the ontology class representing its type. Depending on the used semantic web services framework a suitable language for ontologies definitions can be selected. Since we use web services described in OWL-S which is based on the OWL language, we assume the use of OWL for representation of ontologies in the further text.

We developed an ontology[1] of event types specific to OWL-S web services execution with each particular event type represented by one OWL class. Figure 1 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, *pro-*

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

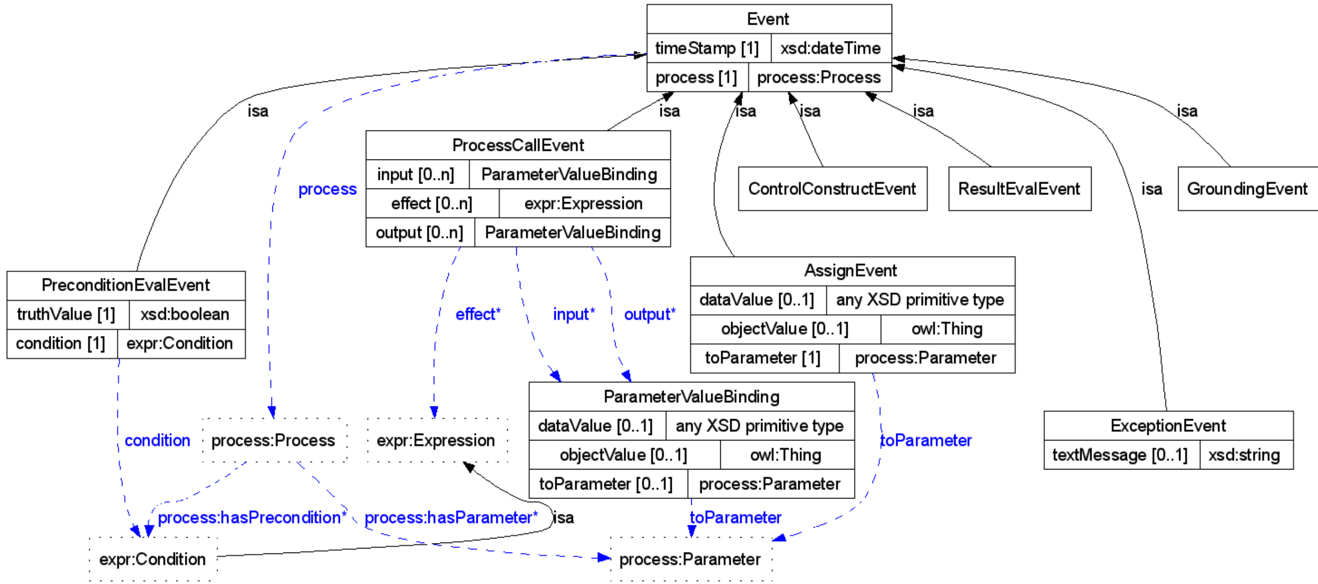

Figure 1. Direct subclasses of the Event class with their properties

cess:Parameter means that the *Parameter* class is defined in the OWL-S process ontology.

The *Event* class is a common super-class of all event types. 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. The following list summarizes event types and corresponding classes in the events ontology that can occur during the execution of the process model:

- **Process call (the *ProcessCallEvent* class and its subclasses):** For each process type (i.e., atomic, composite and simple) specific event types are defined representing its start and end. 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. Figure 2 shows an example event with inputs and outputs assigned.
- **Parameter assignment:** The *AssignEvent* and its subclasses represent inputs assignments and outputs processing.
- **Preconditions evaluation:** 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).
- **Result evaluation:** The *ResultEvalEvent* and its subclasses represent evaluation of a conditional result comprising produced effects and output bindings.
- **Control construct execution (the *ControlConstructEvent* and its subclasses):** 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.
- **Grounding events:** The *GroundingEvent* and its subclasses represent events that can occur during WSDL grounding processing.
- **Failures and erroneous events (represented by the *ExceptionEvent* and its subclasses):** For different categories of errors specific event types are defined. The *ExceptionEvent* defines a *textMessage* property containing a text message with detail information about the exception.

More detailed description of the OWL-S events ontology is provided in [19].

Example 1: Figure 2 shows an instance of one event emitted by the OVM during the execution of the *Login* service introduced in Section 3. The *shoppingService* namespace refers to the process model of the *Login* web service. In this example, the event refers to the end of the execution of the *&shoppingService;Login* atomic process with "John" as the value of the *&shoppingService;username* input parameter, and "secret word" as the value of the *&shoppingService;password* input parameter. The service returned "true" as the value of the *&shoppingService;status* output parameter.

```
<AtomicProcessEndEvent>
  <timestamp>2007-03-12T12:35:12</timestamp>
  <process rdf:resource="&shoppingService;Login"/>
  <input>
    <ParameterValueBinding>
      <toParameter
          rdf:resource="&shoppingService;username"/>
      <dataValue>John</dataValue>
    </ParameterValueBinding>
  </input>
  <input>
    <ParameterValueBinding>
      <toParameter
          rdf:resource="&shoppingService;password"/>
      <dataValue>secret word</dataValue>
    </ParameterValueBinding>
  </input>
  <output>
    <ParameterValueBinding>
      <toParameter
         rdf:resource="&shoppingService;status"/>
      <dataValue>true</dataValue>
    </ParameterValueBinding>
 </output>
</AtomicProcessEndEvent>
```

Figure 2. An event instance: atomic process call end event representing successful *Login* service call

Operator	Explanation
$A \wedge B$	Conjunction. Occurs when both A and B occur irrespective of their order.
$A \vee B$	Disjunction. Occurs when A or B occurs.
$A; B$	Sequence. Occurs when A occurs before B.
$A - B$	Negation. Occurs when there is an occurrence of A during which there is no occurrence of B.
A_t	Temporal restriction. Occurs when there is an occurrence of A shorter then t time units.
$A + t$	Temporal event. A temporal event is a special type of a primitive event that occurs t time units after an occurrence of A. A temporal event occurrence refers to the event occurrence of A that initiated it.

Table 2. Composition operators

5 Event detection algebra

Event detection algebras present a mechanism for composite events specification and detection. In event algebras, primitive events and a number of operators are used to form event expressions that represent an event pattern of interest. Event algebra allows us to combine primitive event types into composite event expressions. In this section we describe operators and semantics of an event algebra defined in [3]. For simplicity we assume a discrete time model with abstract time units represented by natural numbers. T is used to denote the time domain. In an implementation, real time units as seconds and minutes are used instead.

As we mentioned before, by *primitive event occurrence* we mean an instantaneous, atomic occurrence of an interest at a point in time which is characterized by some *event type* and data values in the form of parameters associated with it. In this section, parameters will not be considered. We also assume that the set of event types is predefined by the system (as, e.g., defined in the previous section).

Let $\mathcal{P}$ denote a finite set of event type identifiers that are of interest to the system. For each event type $E \in \mathcal{P}$ $dom(E)$ denotes the set of *primitive event instances* (or primitive events) of type E. For a primitive event e we say that it is an instance of the event type E if and only if $e \in dom(E)$. A *primitive event occurrence* is represented as a singleton set of the form $\{< E, e, t >\}$, where $< E, e, t >$ is a triple, E is an event type, e is a primitive event instance of that type and t is a time when the event occurred (t corresponds to the *timeStamp* value of the *Event* class as defined in Section 4). A set representation allows us to treat primitive and composite event occurrences uniformly.

During the system execution primitive event occurrences are emitted. At a given time, several occurrences of different event type can be emitted, however simultaneous occurrences of the same primitive event type are not allowed. Event occurrences form *event streams*. A *primitive event stream* is a set of primitive event occurrences of the same event type with different times. We use event streams to define the event algebra semantics. A particular scenario (or an execution in our case) can be described be means of event streams. We introduce an *interpretation* as a way of describing one particular scenario, i.e., the occurrences of primitive events. Formally, an *interpretation* is a function mapping each event type $E \in \mathcal{P}$ to a primitive event stream containing primitive event occurrences of the type E.

Composite events are defined by *event expressions* build from primitive event types and algebra operators. Table 2 introduces event composition operators. Event expressions are defined inductively. For each $A \in \mathcal{P}$, A is an event expression. If A and B are event expressions, and $t \in T$, then $A \wedge B$, $A \vee B$, $A; B$, $A - B$, A_t, $A + t$ are event expressions.

With event expressions introduced we can extend the notion of event occurrences and event streams also to composite events. An *event occurrence* is a union of n primitive event occurrences, where $n > 0$. For an event expression A, a *composite event occurrence* is represented by primitive event occurrences that caused the occurrence of the event described by the expression A. Since composite events are not instantaneous but stretch over some interval, the occurrence interval for a composite event occurrence needs to be defined. For an event occurrence e we define $\mathrm{start}(e) = \min(\{t| < E, v, t > \in e\})$ and $\mathrm{end}(e) = \max(\{t| < E, v, t > \in e\})$.

Formally, occurrences of composite events are derived from the definition of event streams. A *general event stream* is a set of event occurrences. The following composition functions over streams are used to define semantics of composition operators. If S and T are event streams and $t \in T$, define:

$$\begin{aligned}
\mathrm{dis}(S,T) &= S \cup T \\
\mathrm{con}(S,T) &= \{s \cup t \,|\, s \in S \wedge t \in T\} \\
\mathrm{neg}(S,T) &= \{s \,|\, s \in S \wedge \neg\exists t(t \in T \wedge \mathrm{start}(s) \leq \\
&\quad \leq \mathrm{start}(t) \leq \mathrm{end}(t) \leq \mathrm{end}(s))\} \\
\mathrm{seq}(S,T) &= \{s \cup t \,|\, s \in S \wedge t \in T \wedge \mathrm{end}(s) \leq \mathrm{start}(t)\} \\
\mathrm{tim}(S,t) &= \{s \,|\, s \in S \wedge \mathrm{end}(s) - \mathrm{start}(s) \leq t\} \\
\mathrm{delay}(S,t) &= \{r \,|\, s \in S \wedge r \,\text{is a new temporal event} \\
&\quad \text{refering to}\, s \wedge \mathrm{start}(r) \leftarrow \mathrm{end}(r) \leftarrow \mathrm{end}(s) + t\}
\end{aligned}$$

Using defined stream composition functions a semantics of operators can be defined. For an interpretation function $\mathcal{I}$ the semantics of event expressions is defined as follows:

$$\begin{aligned}
[A]^{\mathcal{I}} &= \mathcal{I}(A) \text{ if } A \in \mathcal{P} \\
[A \vee B]^{\mathcal{I}} &= \mathrm{dis}([A]^{\mathcal{I}}, [B]^{\mathcal{I}}) \\
[A \wedge B]^{\mathcal{I}} &= \mathrm{con}([A]^{\mathcal{I}}, [B]^{\mathcal{I}}) \\
[A - B]^{\mathcal{I}} &= \mathrm{neg}([A]^{\mathcal{I}}, [B]^{\mathcal{I}}) \\
[A; B]^{\mathcal{I}} &= \mathrm{seq}([A]^{\mathcal{I}}, [B]^{\mathcal{I}}) \\
[A_t]^{\mathcal{I}} &= \mathrm{tim}([A]^{\mathcal{I}}, t) \\
[A + t]^{\mathcal{I}} &= \mathrm{delay}([A]^{\mathcal{I}}, t)
\end{aligned}$$

These definitions result in an algebra with a simple semantics and intuitive algebraic properties (e.g., commutativity of $\wedge$, $\vee$ operators, associativity of $\wedge$, $\vee$, ; and distributivity of $\wedge$, $\vee$). For more details see [3]. According to this definition, given an interpretation function $\mathcal{I}$, each event expression identifies an event stream with all composite event occurrences that match the given expression.

6 Semantic filtering

In this section, we extend the detection algebra by introducing events filtering based on expressions in the form of conjunction of description logics atoms enriched with OWL datatypes and SWRL built-ins. The format of filtering expressions is motivated by SWRL [8] expressions that are used in SWRL rules antecedents. Such expressions allow us to match events represented as OWL instances. We assume existence of a knowledge base KB that is used for evaluation of filter expressions. Specifically, an execution engine (the OVM) maintains the KB during execution of the process model and stores produced results in the KB. Also, the KB contains definitions introduced in the executed OWL-S process model.

A *filter expression* is a conjunction of atoms. An *atom* can be one of the following expressions: $C(s)$ (concept atom), $Po(s,t)$ (object property atom), $Pd(s,d)$ (datatype property atom), $sameAs(s,t)$ (same as atom), $differentFrom(s,t)$ (different from atom) and $builtinID(d_1, ..., d_n)$ (built-in atom), where C is an OWL class name (primitive event type), Po is an OWL object property, Pd is an OWL datatype property, $builtinID$ is an identifier of some SWRL built-in predicate with arity n, $d_1, ..., d_n$ are variables or OWL data values, s and t are variables or OWL individuals in the KB and d is a variable or an OWL data value.

A filter expression holds with respect to the KB, if there exists an assignment of individuals (from the KB) and data values to all variables in the expression, such that all atoms hold. Informally, an atom $C(s)$ holds if s is an instance of the class C, an atom $Po(s,t)$ holds if s is related to t by property Po, an atom $Pd(s,d)$ holds if s is related to d by property Pd, an atom $sameAs(s,t)$ holds if s is interpreted as the same object as t, an atom $differentFrom(s,t)$ holds if s and t are interpreted as different objects, and $builtinID(d_1, ..., d_n)$ holds if the built-in relation $builtinID$ holds on the interpretations of the arguments (see [8] Sections 3 and 8).

In general, we allow every event expression to be associated with a filter expression. However, to maintain control over event detection we also introduce a restricted form of event expressions in which filter expressions can be associated with primitive event types only.

First, we introduce restricted event types. A *restricted event type* is defined as follows: $T = ?v : A[F]$, where T is a name of the defined event type, $A \in \mathcal{P}$ is a primitive type, $?v$ is a variable[2] which is used to refer to a primitive event occurrence that is detected as an instance of A, and F is a filter expression. The interpretation of restricted event types is straightforward: during event detection for every event occurrence of type A the filter expression F is evaluated and only if it holds the event occurrence is detected also as an instance of T. A variable $?v$ can be used in the expression F to refer to detected event occurrence.

Example 2: The following expression defines a new restricted event type *Logout* derived from the $AtomicProcessEndEvent$ event type:
Logout $= ?x : AtomicProcessEndEvent[$
$process(?x, ?process) \wedge$
$sameAs(?process, "\&shoppingService; Logout")]$
An occurrence of the *Logout* event type will be detected only when an atomic process identified by the *&shoppingService;Logout* URI finishes its execution and an appropriate *AtomicProcessEndEvent* event is emitted. A *process* property in this example is the property defined for every event instance (see Figure 1 and 2).

We define *restricted event expressions* as event expressions in which defined restricted types can be used in the

[2] Variable names in expressions are marked with a ? sign, e.g., $?x$.

same fashion as primitive types.

Additionally, we define *extended event expressions* as event expressions in which filter expression can be attached to any event subexpression. If A is a valid event expression, also $A[F]$ and $?v : A[F]$ are valid expressions. F stands for a filter expression and $?v$ is a variable identifying an event occurrence which was detected as an instance of A. In extended event expressions, restricted event types can be used as well.

Obviously, extended event expressions offer more expressive power than restricted expressions, since filters attached to any subexpression allow to define conditions depending on several primitive type occurrences. On the other hand, restricted event expressions are more suitable for runtime event detection, because more efficient detection mechanisms can be used, as we show in the next section. Also note that since in restricted event expressions filter expressions are "hidden" in restricted event type definitions, the semantics of composition operators and the properties of the event algebra remain unchanged. With extended expressions one must be more careful, especially during detection of composite events, as we also show in the next section.

The following examples use web services definitions introduced in Section 3. In the examples we assume that restricted types $Login$ and Buy were defined in a similar fashion as the $Logout$ type in Example 2.

Example 3: The following restricted event expression can be used to detect situations when a customer logs out without buying anything (event pattern 2b from Section 3):

$((Login; Logout) - Buy)$

Example 4: The following extended event expression detects 2 *Login* calls of the same user within an interval of 120 seconds. It is a simplified version of the problem 2a in Section 3:

$(?log1 : Login; ?log2 : Login)_{120}$ [
input(?log1, ?par1) $\wedge$ *toParameter(?par1, ?par1Name)* $\wedge$ *sameAs(?par1Name, "&shoppingService;username")* $\wedge$ *dataValue(?par1, ?userName1)* $\wedge$ *input(?log2, ?par2)* $\wedge$ *toParameter(?par2, ?par2Name)* $\wedge$ *sameAs(?par2Name, "&shoppingService;username")* $\wedge$ *dataValue(?par2, ?userName1)*]

Properties *input*, *toParameter* and *dataValue* are defined in the events ontology for corresponding OWL classes (for an example of an event instance see Fig. 2).

7 Events detection

For primitive event occurrences represented as instances of OWL classes, primitive event detection translates into finding a set of event types for which a given event occurrence belongs into their domains. Formally, let $e = \{< E, v, t >\}$ be an event occurrence of an event type E, and A be an event expression of interest. Let $candidateTypes = \{A_1, ..., A_n\}$ denote a set of event types occurring in A. To identify all event types that must be notified when e occurs, we need to find a set

$typesToNotify = \{C \,|\, C \in candidateTypes \wedge C \text{ superclass of } E\}$,

since we know that for each type C if C is superclass of E and $e \in \text{dom}(E)$ then $e \in \text{dom}(C)$.

For every member of the $typesToNotify$ set, e will be added into its primitive event stream. Finding the $typesToNotify$ is actually very easy. The only thing we need to compute is a set of superclasses for each event type. As an optimization, this set can be computed during the system initialization.

For a restricted event type $T = ?x : A[F]$ with filter expression F, additionally the filter expression needs to be evaluated after a primitive event occurrence *e* of type *A* is detected to decide, if e should be added to the primitive event stream of T as well.

7.1 Composite events detection

In earlier works, various mechanisms for composite event detection were proposed, including Petri nets, finite state automata, and event detection graphs. We adopted an approach based on event detection trees presented in [4], however we modified the algorithm according to the work in [3] which detects sequences of events correctly, and extended it with filtering expressions. Before giving an overview of the detection method, we first need to introduce a concept of restriction policies.

An occurrence of a composite event is caused by occurrences of primitive events that match a given event expression. Since primitive events can occur repeatedly, several possible combinations of primitive event occurrences can trigger a composite event at a certain time. Figure 3 illustrates such a situation. The upper part of the figure shows primitive event streams for event types *LookupItem* and *AddItemToCart* displayed on the time-lines a and b running from left to right. Black boxes stand for primitive event occurrences. For example, instances of *LookupItem* type occur at times 1, 2 and 4. The bottom part of the figure illustrates two different detection regimes of an event expression *(LookupItem;AddItemToCart)*. Each composite event occurrence is displayed as a pair of white boxes connected by a dotted line. Time-line c displays all occurrences of composite event gained as all possible combinations of primitive events. For example, at time 3, two different instances of a composite event occur. Generally, due to the combinatorial explosion too many occurrences need to be detected, especially when primitive events can occur repeatedly. In many situations, such a solution is not desirable and even not possible because of time and memory constraints.

In [4] authors defined several possible policies that restrict the amount of detected composite events to deal with

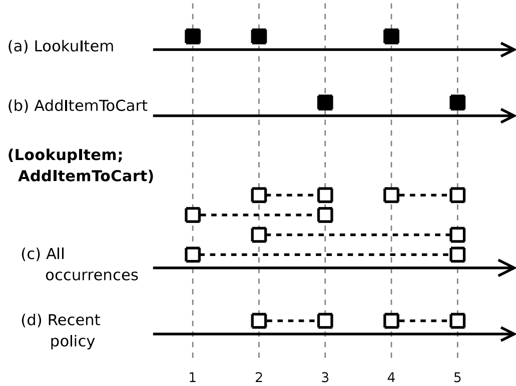

Figure 3. Composite events detection

combinatorial explosion. We will focus on the *recent policy*[3] which maintains an invariant that at each time at most one event occurrence for a given event expression is detected. If there are more candidates, the one with the maximal start time is detected. In Figure 3, event occurrences detected by applying the recent policy are shown on the time-line d.

Formally, if S is an event stream and S' is an event stream which we get from S by applying the recent policy to it, the following holds:

1. $S' \subseteq S$
2. $\forall s\,(s \in S \Rightarrow \exists s'(s' \in S' \wedge \mathrm{start}(s) \leq \mathrm{start}(s')))$
3. $\forall s, s'\,((s \in S' \wedge s' \in S' \wedge \mathrm{end}(s) = \mathrm{end}(s')) \Rightarrow s = s')$

Such a policy guarantees that if there is one or more event occurrences of a composite event, one of them will be always detected, which is a property required in many applications contexts.

It is shown in [3] that when the recent restriction policy is applied to the event algebra based on event streams, we still get an algebra with intuitive properties. Additionally, event detection adhering to the recent policy can be implemented efficiently, since only the most recent primitive event occurrence of each primitive event type needs to be remembered. This makes the recent policy detection suitable especially for runtime detection of composite events.

Since, as we noted in the previous section, allowing restricted types (i.e., event types with filter expressions) in event expressions does not have any effect on properties of the event algebra, the same detection mechanisms using the recent policy can be applied for detection of restricted event expressions. However, this is not true for extended event expressions. Specifically, the property 2 guaranteed by the recent policy does not always hold for event streams of extended event expressions. To demonstrate this, we attach a filter F to the event expression *(LookupItem;AddItemToCart)* so that we get an extended expression. Now, let us consider for example the situation at time 3 in Figure 3. Before applying the filter F, in general case (time-line c), there are two candidate composite event occurrences that can be detected. If the expression F holds only for the one with the earlier start and does not hold for the one the later start, detection using recent policy will detect no composite event at time 3, while in an unrestricted case a composite event is detected, which is in contradiction with property 2 of the recent policy. This means, that either the recent policy should be used only for detection of restricted event expressions, or when used with extended event expressions, some composite events occurrences can be missed.

In our system, we use the recent policy and restricted event expressions for detection of complex event occurrences during the runtime execution of the web service process model. For the off-line after-execution analysis of recorded event streams extended event expressions with an unrestricted detection mechanism can be used, since in the off-line case the expression power is more important than the resource efficiency. Other

7.2 Event detection trees

Event detection trees [4] present an efficient mechanism for detection of composite events defined by event expressions. For detection purposes, each event expression is represented as a tree with leaves representing event types occurring in the expression and every other node representing one composition operator in the expression. The tree represents a decomposition of the event expression into its subexpressions, starting from the root standing for the whole expression and ending in leaves representing event types. Detection of composite event occurrences starts at the bottom with detecting primitive event occurrences and proceeds in the bottom up direction by progressively detecting occurrences of more complex subexpression until eventually the root is reached and an event occurrence for whole expression is detected. Whenever a new event occurrence is detected by some node, the node notifies its parent so that the parent node can test if the newly detected event occurrence reported by the child causes an occurrence of a more complex event matching the subexpression of the parent. Every node maintains a history of event occurrences in its own buffer. However, when the tree implements the recent policy, only last event occurrence needs to be remembered at each node (with the only exception for the ; operator. For details see [3]).

Figure 4 shows two detection trees for event expressions $(LookupItem; AddItemToCart)$ and $((LookupItem \wedge AddItemToCart); BuyAndShipItems)$. We assume

[3]The recent policy is suitable especially for run-time event detection. For examples of other policies suitable for different purposes we refer to [4].

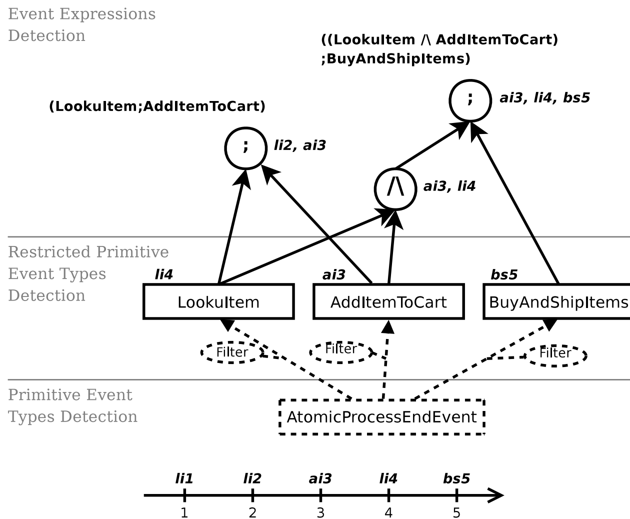

Figure 4. Event detection trees

that restricted event types *LookupItem*, *AddItemToCart* and *BuyAndShipItems* are defined in a similar fashion as in Example 2. Please note that since two trees are displayed and since these trees share some event types, we allowed nodes representing event types to have more than one parent node. This anomaly breaks the strict tree definition, but it does not have any negative effect on the detection method. The only difference is that whenever a primitive event occurrence is detected, the appropriate leave node notifies all its parents. We extended the original concept of detection trees by adding support for restricted types detection and for event filters.

The time-line at the very bottom of the Figure 4 shows primitive event occurrences emitted by the system during the process model execution. Event occurrences *li1*, *li2* and *li4*, emitted at times 1, 2 and 4 respectively, are instances of the *LookupItem* restricted type. *ai3* is an instance of *AddItemToCart* and *bs5* is an instance of *BuyAndShipItems*. In addition to the static structure of detection trees, the figure represents a snapshot of the detection process at time 5. Current values of detected events for each subexpression are displayed next to each node (printed in bold italics).

The figure is logically divided into three layers. The bottom layer shows detectors of event types, the middle layer displays nodes responsible for detection of restricted event types, and the top layer displays detection trees and nodes realizing composite event detection. Since in this particular case all three restricted event types are derived from the *AtomicProcessEndEvent* event type, the bottom layer contains only one node for detection of a event type (shown as a dotted box). When a event occurrence of the *AtomicProcessEndEvent* event type is detected, the node notifies all its parents (dotted arrows). If the parent represents a restricted event type (as in our case), it evaluates its associated filter expression to test if the reported event occurrence matches it.

For example, at time 5, *bs5* occurrence is detected by the *AtomicProcessEndEvent*. From the three parents of *AtomicProcessEndEvent*, only *BuyAndShipItems* updates its state since filters of other two restricted event types do not hold for *bs5*. Because the state of *BuyAndShipItems* node has changed, it notifies its parent, which represents the ; operator of the $((LookupItem \wedge AddItemToCart); BuyAndShipItems)$ expression (the root node). After receiving notification about *bs5*, the root node (labeled with ;) combines it with its previous state (instance $ai3, li4$ was already detected at time 4 for the $LookupItem \wedge AddItemToCart$ subexpression) and identifies a new occurrence $ai3, li4, bs5$. Since at this point the root of the tree was reached, an occurrence of a new composite event is reported to the system.

For details on implementation of operator nodes adhering to the recent policy please refer to [3]. Described mechanism is used during the runtime detection of restricted event expressions. To support detection of extended expressions each operator node additionally uses a filter (not displayed in the Figure 4) of the corresponding subexpression to decide whether a candidate event instance matches the filter or not.

A final note of this section is related to filters evaluation. An expression language introduced in filters is quite complex and evaluation of filters with respect to the knowledge base KB maintained by the process model execution component is not trivial. In our system, first bound variables are replaced by their values. Then the filter expression is translated into a SPARQL [1] query that searches for values of free variable satisfying the filter expression. Finally, the query is evaluated w.r.t. the KB and if some result is return, the filter expression holds, otherwise it fails. Since during the runtime detection the KB contains only instances produced within this session, i.e. the size of KB is relatively small, the query can be evaluated efficiently.

8 Related work

Event algebras were developed originally in the context of active databases [4, 6] and later used in distributed [11, 15], real-time [12] and embedded systems [3]. Different aspects of events monitoring, as predictable resource requirements, detection efficiency or delayed events occurrences in distributed systems were addressed. Detection mechanisms include Petri nets [6], event graphs [4] and finite state automata [15].

Extensive work has been done in the area of events processing, passing and monitoring. A general coverage of events-based systems is provided in [10]. Distributed mid-

dleware 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. Such systems are usually concerned with monitoring of performance, availability and other SLA metrics. The Web Service Level Agreement (WSLA) framework [9] is targeted at defining and monitoring SLAs for Web Services. Sahai et. al. [17] developed an automated and distributed SLA monitoring engine that allows definition of SLAs and their automatic monitoring and enforcement. The general problem of current systems is the lack of machine processable semantics as identified in [7]. The only work know to us that addresses the problem of semantic process monitoring is presented in [14]. An ontology for process monitoring and mining is used in the context of the Super project that builds on WSMO framework [16].

9 Conclusions

In this paper, we described primitive and composite event specification and detection mechanisms suitable for monitoring of semantic web services. The proposed approach, which we call semantic monitoring, combines semantically rich primitive events with an event algebra used for composite events specification. We augmented the event algebra with semantic filtering that allows occurred events to be matched against specified filtering expressions, and we also described appropriate detection mechanisms. Further, we identified a restricted variant of the event algebra combined with semantic filters that is suitable for runtime monitoring. Although, our work is based on OWL-S, the proposed language for specification of composite semantic events can be easily adapted and used with different semantic web services frameworks, such as WSMO.

In the future work we want to study questions related to the efficiency and scalability of the proposed approach. Especially in case of large, complex process models and executions stretching over a long period, the KB base can grow substantially which might negatively impact efficiency of event expressions evaluation. A solution to this problem might be in imposing more strict assumptions on the detection algorithm, for example, in the form of an explicit time limit for events lifetime as proposed in [12].

References

[1] Sparql query language for rdf. W3C Candidate Recommendation 14 June 2007, http://www.w3.org/TR/rdf-sparql-query/.

[2] 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/.

[3] J. Carlson and B. Lisper. An event detection algebra for reactive systems. In G. C. Buttazzo, editor, *EMSOFT*, pages 147–154. ACM, 2004.

[4] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 606–617, Santiago, Chile, 1994.

[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] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Rules in Database Systems*, pages 23–39, 1993.

[7] 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.

[8] I. Horrocks, P. F. Patel-Schneider, H. Boley, B. Grosof, and M. Dean. Swrl: A semantic web rule language combining owl and ruleml. http://www.w3.org/Submission/SWRL/.

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

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

[11] M. Mansouri-Samani and M. Sloman. GEM: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

[12] J. Mellin. *Resource-Predictable and Efficient Monitoring of Events*. PhD thesis, Computer Science School of Humanities and Informatics University of Skovde, 2004.

[13] 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.

[14] 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.

[15] P. R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *IEEE Network*, 18(1):44–55, 2004.

[16] 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.

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

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

[19] R. Vaculín and K. Sycara. Monitoring execution of OWL-S web services. In *European Semantic Web Conference, OWL-S: Experiences and Directions Workshop*, June 3-7 2007.