ABSTRACT
Monitoring of business processes and service-oriented systems is a critical enabling technology for improving visibility into business operations, allowing their optimization, adjustments and restructuring. One of the major problems of current monitoring methods is the prevalent heterogeneity of various types existing among the applications and services used in complex distributed business process. In this chapter we propose semantic monitoring as a possible solution addressing some of the heterogeneity problems. The idea of semantic monitoring is to apply semantic annotations, using ontologies, to descriptions of event types and event instances emitted during interactions with integrated applications. We introduce a generic semantic monitoring framework consisting of a modular monitoring ontology and appropriate event detection mechanisms. The monitoring ontology defines generic, language independent monitoring concepts, and the language specific modules defining taxonomy of event types specific to a particular process modeling language/methodology. We present two such modules, one we have developed for the OWL-S Semantic Web services process models, and the other that we have developed for a business artifact-centric approach to business process specification. Next, we describe mechanisms for specification and detection of semantic 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. 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.

INTRODUCTION
The growing complexity and high adoption of business processes lead to increased needs for methods improving their understanding and visibility, allowing optimization and adjustments of business operations. In the broad sense, monitoring is recognized as a natural part of Business Process Management (BPM) systems and the Service-Oriented Architectures (SOA). In the context of BPM and Business Process Analysis (BPA) (van der Aalst et al., 2003), monitoring enables insights and understanding of the BPM and SOA systems, supporting business intelligence, optimization or business re-engineering methods. Importantly, BPM is approaching the management and execution of IT-supported business operations from a business expert’s view rather than from a technical perspective (Smith and Fingar, 2003) and therefore the monitoring methods need to provide an appropriate abstraction over the technical notions of underlying IT systems, often realized as (web) services.

Despite a significant body of work in the business processes monitoring (e.g., Costello and Molloy, 2009; Jeng et al., 2004; Thomas et al., 2005; Lakshmanan et al., 2010) developing
effective and efficient monitoring models and methods remains a hard and laborious problem with many challenges (Corea and Watters, 2007). One of the major reasons why the existing methods are not perfectly suitable for monitoring of distributed, rapidly changing business processes are the prevalent heterogeneities of various types that exist among the applications and services used in business processes — such as process, data, documents, vocabulary, or schemas heterogeneity. Typically such applications and services are developed and maintained by several independent vendors, oftentimes the processes cross organizational boundaries, and similarly often the processes need to integrate applications that have been developed in isolation and which reside in multiple isolated application silos within an organization. Such heterogeneities present a substantial problem for the monitoring infrastructures which typically need to provide a unified view across the business domain, so as to appropriately support applications such as business intelligence, business analytics, or business optimization and transformation. Similarly, powerful unified monitoring infrastructure is needed for realizing the vision of reliable, adaptive business processes operating in dynamic environments.

In this chapter we will discuss various aspects of semantic monitoring, which we propose as a possible solution addressing some of the above mentioned heterogeneity problems, and which brings some substantial benefits for business processes monitoring, compared to traditional monitoring methods. The idea of semantic monitoring is to apply semantic annotations — using ontologies (Baader et al., 2003) — to descriptions of event types and event instances emitted during interactions with integrated applications (e.g., in the form of Semantic Web services). Similarly to traditional monitoring approaches, we propose to organize event types in a taxonomy. However, in the semantic monitoring approach we define event types taxonomy and event attributes as formal ontology concepts (we use the terms concept and class interchangeably in the chapter assuming the same meaning for both). Additionally, data associated with an event instance are annotated by ontology concepts.

Using ontologies for annotations of events and for their definition has several advantages. Most importantly, the shared ontologies can serve as an integrating semantic layer (or overlay) over heterogeneous applications. Further, due to a clearly defined semantics and standardized serializations of ontologies (e.g., the OWL standard defines a serialization into XML syntax), events and their content can be easily processed and shared by software agents and various applications. Next, instead of pure syntactic matching during event detection, semantic reasoning can be used to support more flexible event detection. Finally, after the service execution is finished, complex filtering and querying techniques exploiting the rich semantic interaction trace can be used for post-execution analysis.

We introduce a generic semantic monitoring framework consisting of a modular monitoring ontology and appropriate event detection mechanisms. The monitoring ontology consists of a generic module which defines monitoring concepts that are independent of a particular process modeling language. Additionally, our monitoring ontology has modules specific to a particular process modeling language/methodology. We will present two such process language specific monitoring ontology modules, one that we have developed for the OWL-S (Martin et al., 2004) Semantic Web services process models, and the other that we have developed for a business
artifact-centric approach (Nigam and Caswell, 2003; Bhattacharya et al., 2007; Chao et al., 2009) to business process specification. Both, the Semantic Web services approach and the business artifact-centric approach are in particular attractive for the business monitoring purposes since they provide a unifying abstraction and a layer over the underlying, possibly heterogeneous monitored applications and processes. For example, the business artifacts methodology is often used as a technique for horizontal integration of heterogeneous applications.

For many applications detection of individual events emitted by various components of the systems is a sufficient solution. However, often complex event patterns (called composite events) need to be detected. We have developed a language for specification of semantic composite events based on the event algebra introduced originally in the context of active databases (Chakravarthy et al., 1994). Finally, we propose monitoring mechanisms suitable for both efficient runtime monitoring and offline analysis. We adopted an approach based on event detection trees proposed by Chakravarthy et al. (1994), however we modified the algorithm according to the work of Carlson and Lisper (2004) which presents a more efficient technique suitable especially for runtime detection. We extended the algorithm so as to allow detection of semantic events.

The chapter is organized as follows. We start with providing a brief discussion of background and related work. Next, we describe benefits of semantic monitoring and introduce example problems that can be addressed by semantic monitoring and event detection. In the following section we define the core concepts of our technical approach, such as primitive events, and we introduce the monitoring ontology, defining a small set of independent event types, and two sets of event types specific to OWL-S Web services approach, and to business artifact-centric approach. Next we will discuss mechanisms for definition and detection of composite event patterns suitable in the context of semantic monitoring, based on the event detection algebra which we extended with semantic event filtering to support detection of events with semantically rich content. Finally, we will conclude the chapter with a brief discussion and with directions for future work.

BACKGROUND AND RELATED WORK
In complex business processes and workflows, the needs for analyzing, diagnosing, simulating and optimizing arise. Monitoring systems can be seen as enablers of these functionalities, and as such the monitoring systems need to support

- measuring of (runtime) metrics and key performance indicators, including support for notifications and auditing of service performance,
- measuring and evaluation of Quality of Services metrics (QoS),
- enforcement of Service Level Agreements (SLA), including 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.

With respect to monitoring, in service-oriented and BPM systems traditionally the main stress is put on monitoring of performance and availability metrics. The monitoring subsystems typically
address problems such as measuring and evaluation of Quality of Services (QoS) metrics, enforcement of Service Level Agreements (SLA), alert-based reporting on the level of adherence to the SLA, automated sending of notifications related to service performance, performance auditing, etc. For example, the Web Service Level Agreement (WSLA) framework (Keller and Ludwig, 2003) is targeted at defining and monitoring SLAs for Web Services. The WSLA framework consists of a flexible and extensible language based on XML Schema and a runtime architecture comprising several SLA monitoring services. SLA monitoring problems in multi-provider environments were considered by Overton in (Overton and Siegel, 2002) and (Overton, 2002). Sahai et al. (2002) developed an automated, distributed SLA monitoring engine that allows definition of SLAs and their automatic monitoring and enforcement.

Similarly to Web service based systems, distributed middleware systems based on CORBA or JMS typically include monitoring subsystems and tools for analyzing logged events. Such systems are typically concerned also with monitoring of performance, availability and other SLA metrics.

The aforementioned existing systems and tools usually use different formats for reading/storing log records and present their results in a different fashion. This problem is addressed by the ProM framework (pluggable environment for process mining) (van Dongen et al., 2005). The goal of the ProM framework is to define independent algorithms for process mining. ProM uses a generic format for representation 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 by Hepp et al. (2005), where a vision of using Semantic Web services for Semantic Process Management is presented. de Medeiros et al. (2007) further give an outlook on the opportunities and challenges of semantic business process mining and monitoring. The problem of lacking machine processable semantics in process monitoring and analysis is addressed by Pedrinaci et al. (2008). The authors present a generic ontology for process monitoring and process mining. The ontology was developed in the context of the Super project that builds on the WSMO framework (Roman et al., 2005). The ontology builds upon a Time Ontology and is structured around the process, resource, and object perspectives as typically adopted when analysing business processes. The ontology is similar to our approach, except we base our work on the OWL-S ontology, while Pedrinaci et. al. base the ontology on the WSMO framework.

A different approach to monitoring of BPM systems is presented by Barros et al. (2007). The authors extend the BPEL and BPMN languages with complex event patterns as they are known in the field of Complex Event Processing (Luckham, 2002). Decker et al. (2007) propose a graphical notation for modeling composite events in business processes based also focusing primarily on BPMN and BPEL.

An extensive work has been done in the area of event processing, passing and monitoring. A general coverage of events-based systems and complex event processing is provided by Luckham (2002). Luckham presents the Rapide event pattern language which allows a definition of event pattern rules and event pattern constraints. The event model semantics is based on causally related events and events timing, causality, and aggregation are addressed. Muhl et al. (2006)
provide an overview of event-based systems covering topics such as local event matching and distributed event forwarding, composite event detection and security.

Our approach to composite event patterns is based on event algebras which were originally developed in the context of active databases (Chakravarthy et al., 1994; Gatziu and Dittrich, 1993). Later similar approaches were extended and employed in various contexts. Mansouri-Samani and Sloman (1997) and Pietzuch et al. (2004) address the problem of composite event patterns in distributed systems. Mellin (2004) has developed monitoring mechanisms for real-time systems, and Carlson and Lisper (2004) address composite event detection in embedded systems with constrained resources. Different aspects of event monitoring, such as predictable resource requirements, detection efficiency or delayed events occurrences in distributed systems were addressed. Detection mechanisms of complex event patterns include Petri nets (Gatziu and Dittrich, 1993), event graphs (Chakravarthy et al., 1994) and finite state automata (Pietzuch et al., 2004).

Monitoring was also addressed in works dealing with adaptation of workflows (Müller et al., 2004) and business processes (Verma et al., 2006).

**SEMANTIC MONITORING AND ITS BENEFITS**

Most approaches to monitoring are event-driven with events (often from multiple sources) being collected, correlated and analyzed based on predefined event rules or patterns (Luckham, 2002). 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 of primitive events are restricted only to exact event types detection and syntactic attributes matching or comparison. Furthermore, events derived directly from the implementation / definition of heterogeneous applications and business processes will naturally inherit the heterogeneities. Also, often no explicit declarative specification of event types and their attributes is available. Reasons for the lacking semantics in emitted events partly spring from the lack of declarative semantic descriptions of the monitored components itself.

In this work we take inspiration from the area of Semantic Web services. 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 domain ontologies. The idea of semantic monitoring is to apply semantic annotations also to descriptions of event types and event instances emitted during interactions with business processes, Web services and other application components. Similarly to traditional monitoring approaches, we propose to organize event types in a taxonomy. However, in the semantic monitoring approach we define the event types taxonomy and event attributes in an ontology. Additionally, also data items associated with an event instance are annotated by ontology concepts. As discussed in the introduction, such a choice has several advantages, such as a well defined semantics and standardized serializations of ontologies which allow easy sharing and processing of events and their content, and advanced,
flexible events detection based on semantic matching and reasoning.

**Example 1.** In order to demonstrate the benefits of semantic event detection, we introduce a simple semantically annotated process model realizing an electronic shop. The process consists of several Web service tasks (operations) presented in Figure 1 and defined in Table 1. Notice that both the process model and the domain ontologies presented here are simplified and serve for illustration purposes. In particular, we only show the concepts defined in the domain ontology and their IS-A relationships in Figure 1, but we want to stress out that the domain ontology can be any arbitrary ontology with concept definitions, including relationships and constraints, expressed for example in some standard ontology language such as OWL (Bechhofer et al., 2004). In general, consistently with various Semantic Web services frameworks, we assume that processes and/or service operations can be defined in terms of their input and output parameters, and optionally also in terms of their preconditions and effects. For simplicity, in our example we describe only input parameters and a return parameter type (output) for each operation. We assume that all parameter types (e.g., ItemCategory, ItemID, Country, etc.) refer to concepts defined in some domain ontology. The process operates 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 invoked repeatedly. Examples in the following sections will refer to this business process.

![Shopping Process Model](image)

**Figure 1:** Shopping process model including example domain ontologies

In the following paragraphs 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 (process instance) of a given operation/task (e.g., *Logout*).

   (b) Detect a situation where a particular result is produced, e.g., *Login* fails since the username cannot be verified.

   (c) Detect 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 certain time, e.g., 3 unsuccessful *Login* calls within 2 minutes.

   (b) Detect situations where 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 useful during the off-line post-execution analysis:**

   (a) Identify US customers shopping for *Books* who spent more than $1000 within 3 days.

   (b) Analyze popularity of a workflow (specified by some pattern and its features).

   (c) Analyze efficiency of a workflow (e.g., time to buy) or its 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 attributes 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 the required parameter type.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Input names &amp; types</th>
<th>Output name &amp; type</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Login</em></td>
<td>(?username, xsd:string), (?password, xsd:string)</td>
<td>(?status, xsd:boolean)</td>
</tr>
<tr>
<td><em>Logout</em></td>
<td>(?username, xsd:string)</td>
<td>(?status, xsd:boolean)</td>
</tr>
<tr>
<td><em>LookupItem</em></td>
<td>(?category, ItemCategory), (?name, ItemName)</td>
<td>(?item, ItemDetails)</td>
</tr>
<tr>
<td><em>AddItemToCart</em></td>
<td>(?id, ItemID)</td>
<td>(?id, CartId)</td>
</tr>
<tr>
<td><em>BuyAndShipItems</em></td>
<td>(?id, CartID), (?country, Country), (?city, City), (?street, Street)</td>
<td>(?confirmation, CfID)</td>
</tr>
</tbody>
</table>

Table 1: Operations of the shopping process
MONITORING ONTOLOGY AND SEMANTIC EVENTS

In this section we introduce a monitoring ontology and semantic events. First, let us introduce the notion of primitive events. A *primitive event occurrence* is an instantaneous, atomic occurrence of an interest at a point in time (Chakravarthy et al., 1994). Primitive event occurrences are typically directly emitted by the system or its components. Each primitive event occurrence is an instance of some *event type* and it can have additional information in the form of attributes associated with it. In our approach we propose 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. As explained earlier in this chapter, using formal ontologies has various advantages. One of the major advantages is that the shared ontologies can serve as an integrating semantic layer over heterogeneous applications, allowing semantic reasoning and querying of (execution) monitoring logs. The shared ontology layer can be seen as a mediated schema known from traditional database integration area (Halevy, 2001) and it allows to overcome the problem of data and schemas heterogeneity of the underlying monitored applications and processes that are mapped into the mediated schema.

Notice that in some cases the decision to represent event occurrences as instances of ontology concepts (represented for example in the OWL language) is very natural and straightforward. Such cases include all Semantic Web services frameworks (e.g., OWL-S, SA-WSDL, WSMO, SA-REST), or semantic approaches to BPM (Hepp et al., 2005), which directly employ ontologies for semantic annotations of various aspects of the process models. In other cases, where the ontologies are not employed directly (e.g., classical BPMN, WS-BPEL, etc.), a mapping from the underlying applications and their representations to the semantic layer needs to be established by means of well understood mechanisms for lifting of the syntactically represented events to the shared ontology layer. The mechanisms for lifting (i.e., realization of the mappings from the actual system/process representation to the semantic ontology layer) and mechanisms for lowering (i.e., realization of the mappings from the semantic ontology layer back to the system/process representation) help in resolving the heterogeneity problems. In particular, the heterogeneous system and process representations (e.g., data, vocabularies, etc.) can be mapped to the shared ontology layer which provides a consistent and unambiguous vocabulary. Such mappings can be established either manually or semi-automatically and it is out of scope of this chapter to discuss the specific techniques for establishing the lowering and lifting mappings. Also, we do not discuss the problem of *ontology mappings* which is another technique for dealing with heterogeneities and mismatches among ontologies.
Figure 2: Semantic monitoring ontology and its modules

We developed a modular, extensible and domain independent monitoring ontology which defines a taxonomy of event types and their relevant attributes. Figure 2 shows the overall structure of the ontology. The Generic Process Monitoring module defines generic monitoring concepts which are completely independent of the particular process modeling language. This module includes concepts such as a generic event, distinguishing Internal vs. External Events, Composite Event, etc. (we describe those in detail in the following paragraphs). Additionally, we distinguish a Process Language Specific module, which defines event types specific for the particular process modeling language. For various process modeling languages, there will be different modules defining different event types, with the specific event types being derived from the particular language constructs of the particular process modeling language. We will present two such modules, one being a process language specific layer we have developed for the OWL-S, and the other one being specific to the Business Artifact-centric modeling approach. Importantly, when it comes to monitoring of some specific process model of a particular application (such as the one presented in Example 1) the particular content of events will be described in terms of the relevant Domain Ontologies. Such ontologies, in combination with definition of the specific business process (or a family of business processes – e.g., of a given organization unit) might be used to define event types specific for the particular application or application domain (Process Specific Ontology). We provide a mechanism for easy definition of application (domain) specific event types.
Figure 3: Process Monitoring Ontology: Upper right corner shows the generic process monitoring ontology concepts, while the lower left part shows event types specific to OWL-S language (only direct subclasses of the ExecutionEvent class are showed)

Generic Process Monitoring Layer: Taxonomy of Event Types

Figure 3 presents the structure of event types defined in the events ontology with each particular event type represented as one OWL class. We assume the use of OWL language (Bechhofer et al., 2004) for representation of ontologies. In Figure 3 only direct subclasses of the Event class and of the ExecutionEvent class are shown. In the following text we use the Sans serif font for concepts of the events ontology in order to clearly distinguish them from other concepts. Every event type is displayed as a solid box with the name in its heading, and the list of its attributes with cardinalities and the range type specification. Solid arrows with the "isa" label represent sub-classing relation while dashed arrows represent relations between classes. We assume that when an attribute is defined in a particular class, this attribute is “inherited” by all subclasses (thus, for example, every event class has all attributes defined in the Event class). Classes defined in other ontologies are identified by an appropriate namespace and are shown as dotted boxes.

The Event class is a common superclass of all event types. The Event class captures information about generic event instances, such as the possible source of event (application, process, human task, etc.), causal relationships (e.g., what caused the event), context information, and similar. We distinguish four generic subclasses of the Event class. Instances of the ExecutionEvent class represent events that are emitted by the execution infrastructure during execution. Instances of the ExecutionEvent class are associated with a timeStamp referring to the time when the event was emitted. Since an event is always emitted during execution of some process, the process parameter is used to refer to such a process.
The CompositeEvent class is a placeholder for composite events (we discuss composite events in detail in the following section). Instances of the ExternalEvent class represent events that occur outside of the scope of execution and the system gets notified about them — such events might include messages from the user or from other systems. Finally, the ExceptionEvent class is a generic superclass of various types of failures.

**Event Types Specific to OWL-S**

Here we describe a monitoring ontology module that we developed for the OWL-S language. OWL-S (Martin et al., 2004) is a Semantic Web Services description language, expressed in OWL (Bechhofer et al., 2004). Among other things, OWL-S provides constructs for specification of a Process Model which defines how clients can interact with the service by defining the requester-provider interaction protocol. The OWL-S Process Model consists of processes (atomic, composite or simple) that 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. Composite processes are defined by using various control constructs such as sequence, any-order, choice, if-then-else, etc.

The following list summarizes event types specific to OWL-S Web services execution, and corresponding classes in the events ontology characterizing the events that can be emitted during execution of the OWL-S process model:

- **Process call (the ProcessCallEvent class and its subclasses):** For each process type (e.g., atomic, composite and simple) specific event types are defined representing its start and end. The ProcessCallEvent type defines attributes (properties) for specifying input, output and effect values of the executed process. The ParameterValueBinding class used as the range of input and output attributes represents a value assigned to an input or to an output parameter of the process. Figure 5 shows an example event with inputs and outputs assigned (described in Example 2).

- **Parameter assignment:** Instances of the AssignEvent and its subclasses represent an event of inputs assignments and outputs processing.

- **Preconditions evaluation:** Instances of the PreconditionEvalEvent type represent the precondition evaluation occurrences, and refer to the precondition expression with values assigned (the condition attribute) and to the truth value (the truthValue attribute).

- **Result evaluation:** Instances of 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 of OWL-S (such as sequence, split, any-order, etc.) 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:** Instances of the GroundingEvent class 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` attribute containing a text message with detail information about the exception. (Exception types are not discussed in details in this chapter.)

Figure 4 presents a broader view of the taxonomy of event types for OWL-S.

**Example 2.** Figure 5 shows an instance of one event emitted by the execution infrastructure during execution of the LookupItem service introduced in Example 1. Specifically, it shows an instance of the AtomicProcessEndEvent which was emitted after execution of the LookupItem process. The shoppingService namespace refers to the shopping service process model. The event refers to the end of execution of the LookupItem atomic process with “Crime and Punishment” as the value of the name input parameter, and “Novel” (instance of the Books category class) as the value of the category input parameter (typed as Item-Category). The service returned an instance of the ItemDetails class as the value of the item output parameter. Later in this chapter we will illustrate how event instances like this can be used for powerful events detection.
Figure 4: Event Types Taxonomy: Overview of OWL-S process language specific event types
Figure 5: An event instance: atomic process call end event representing a LookupItem service call

It is important to notice that, in two ways, the described event types are application independent. 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. For example, the emitted events can be very easily used for generating logs of the application. If a different monitoring task needs to be performed, such as performance analysis, the same events can be used as well. Thanks to the application independence, the described event types can serve as a sound basis for the monitoring system.
Event Types Specific to Business Artifact-centric Process Modeling

Similarly to OWL-S, event types can be defined for different process modeling languages and approaches. As another illustration we consider a business artifact-centric approach to processes modeling, for which we developed a similar taxonomy of monitoring event types. The business artifact-centric approach (Nigam and Caswell, 2003; Bhattacharya et al., 2007; Chao et al., 2009) considers data as an essential part of business processes, and it defines the business processes in terms of interacting key business artifacts. The business artifacts are usually identified by business users as the key elements of the process and can be seen as a kind of vocabulary of the process (similar to the notion of the domain ontology). Each business artifact (BA) is characterized by an information model and a lifecycle model. The information model records all business-relevant information about a BA instance as it moves through the business. The lifecycle specifies all possible evolutions of a BA instance over time (e.g., in terms of activities/tasks and their flows acting on the data). The BA methodology is often used as a technique for horizontal integration of heterogeneous applications and processes since it provides a unifying view over the heterogeneous domains. This in particular makes the BA approach highly relevant from the point of view of the (semantic) business monitoring.

A BA instance of a particular type is a complex data artifact which can evolve over time in accordance with its lifecycle specification. Various approaches have been considered for representation of BA information models, such as variants of the nested relation model (Thomas and Fischer, 1986), and recently also ontologies have been studied (Bagheri-Hariri et al., 2011) as a suitable representation. Different formalisms have been developed for the lifecycle specification. Most often variations of finite state machines are used, therefore in this chapter we assume the classical FSM (states & transitions) as the formalism for lifecycle specification. We distinguish the notions of tasks and activities. A task represents a logical unit of work. A task may be a human task, an invocation task (e.g., of an external service or some other activity), or an assignment task to/from attributes of the information model. An activity is a named access point that can be invoked to perform some work, and that may specify its operations as flows of tasks.

From the monitoring point of view, the BA information model defines the relevant content (or the business terminology / domain ontology), while the BA lifecycle represents key points in the BA (and process) evolution(s). Therefore, event types can be derived from the major lifecycle constructs so as to capture these important key points. Figure 6 shows the taxonomy of event types defined for the BA approach using the FSM lifecycle. By default, irrespective of the particular event type, the payload of events contains the entire BA information model (depending on the particular purpose only subsets of the information model may be considered or optimizations may be applied – Liu et al. (2011) describe how such optimizations can be defined). This is captured by the BA-ExecutionEvent class, a superclass of all BA monitoring event types (BA-ExecutionEvent type is a direct subclass of ExecutionEvent class as defined in Figure 3).

We distinguish the following major event types:

- Artifact events (instances of the ArtifactEvent class and its subclasses) represent creation
of new BA instances (ArtifactCreated) or changes of information of the BA (DataChanged).

- **State events (instances of the StateEvent and its subclasses)** represent that a BA instance either entered a particular state (StateEntered) or left the state (StateExited).

- **State transition events (instances of the StateTransitionEvent and its subclasses)** represent start (TransitionStarted) and end (TransitionFinished) of a particular state transition in the lifecycle of the BA instance.

- **Activity events (instances of the ActivityEvent and its subclasses)** represent start (ActivityStarted) and end (ActivityFinished) of an activity invocation.

- **Task events (instances of the TaskEvent and its subclasses)** represent start (TaskStarted) and end (TaskFinished) of a task execution.

![Diagram of event types specific to business artifact approach using the FSM for lifecycle representation.]

**Figure 6:** Taxonomy of event types specific to business artifact approach using the FSM for lifecycle representation. BA-ExecutionEvent type is a direct subclass of ExecutionEvent class.

While in this chapter we focus on developing the monitoring ontology and on event detection mechanisms, we want to point out that other aspects of BA monitoring such as employing business artifacts for specification of full fledged monitoring models have been studied elsewhere (Liu et al., 2011).

**EVENT DETECTION ALGEBRA**

Primitive semantic events are useful in various monitoring scenarios. However, as we discussed earlier, in many cases complex event patterns need to be detected. Therefore, we need to provide
a mechanism for definition of complex event patterns which is flexible and which allows an efficient detection. We identified event detection algebras (Chakravarthy et al., 1994) as a suitable formalism that fulfills our requirements. Event detection algebras are a formalism for composite events specification and detection. In event detection algebras, primitive events and a number of operators are used to form event expressions that represent an event pattern of interest. An 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 (Carlson and Lisper, 2004) and in the subsequent section we extend the algebra with semantic filtering which allows detection of semantically annotated events. 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 such as seconds and minutes are used instead.

As we mentioned earlier, 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 attributes associated with the event. We assume that the set of event types is predefined by the system (as described in the previous section). In the following definition we formalize the notion of primitive event instances and their relationship to event types.

**Definition 1 (Primitive Event Instance).** Let $\mathcal{P}$ denotes a finite set of event type identifiers that are of interest to the system. For each event type $E \in \mathcal{P}$ a predicate $\text{dom}(E)$ 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 $e \in \text{dom}(E)$.

Next, we introduce a notion of a primitive event occurrence by associating an event instance with a particular time.

**Definition 2 (Primitive Event Occurrence).** A **primitive event occurrence** is represented as a singleton set of the form $\{\langle E, e, t \rangle \}$, where $\langle E, e, t \rangle$ is a triple with $E$ being an event type, $e$ a primitive event instance of $E$, and $t \in T$ a time when event $e$ occurred.

Primitive event occurrences are emitted during execution. At a given time, several occurrences of different event type can be emitted, however simultaneous occurrences of the same primitive event type are not permitted.

A composite event is a set of primitive event occurrences that are somehow related to each other — for example, there can be a causal relationship.\(^1\) It is the designer of the system or its user who needs to decide which particular sets of event occurrences are of interest. In event algebras, composite events are defined by means of **event expressions** which are constructed from primitive event types and **event algebra operators**. The following definition formalizes the syntax of event expressions.

**Definition 3 (Syntax of Event Expressions).** Event expressions are defined inductively. For each
event type $A \in \mathcal{P}$, $A$ is an event expression. If $A$ and $B$ are event expressions, and $t \in T$, then $A \land B$, $A \lor B$, $A; B$, $A - B$, $A$, $A + t$ are event expressions.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \land B$</td>
<td>Conjunction. Occurs when both $A$ and $B$ occur irrespective of their order.</td>
</tr>
<tr>
<td>$A \lor B$</td>
<td>Disjunction. Occurs when $A$ or $B$ occurs.</td>
</tr>
<tr>
<td>$A; B$</td>
<td>Sequence. Occurs when $A$ occurs before $B$.</td>
</tr>
<tr>
<td>$A - B$</td>
<td>Negation. Occurs when there is an occurrence of $A$ during which there is no occurrence of $B$.</td>
</tr>
<tr>
<td>$A_t$</td>
<td>Temporal restriction. Occurs when there is an occurrence of $A$ shorter than $t$ time units.</td>
</tr>
<tr>
<td>$A + t$</td>
<td>Temporal event. A temporal event is 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.</td>
</tr>
</tbody>
</table>

Table 2: Event expression operators: $A$ and $B$ event expressions, $t \in T$

Table 2 introduces event expression operators and explains their meaning. For example, a conjunction operator $\land$ can be used to create a new event expression $A \land B$ given the existing expressions $A$ and $B$. For example, an expression $\text{AtomicProcessStartEvent} \land \text{AtomicProcessEndEvent}$ represents all occurrences of a composite event that will be identified whenever an occurrence of a primitive event $\text{AtomicProcessStartEvent}$ is detected and an occurrence of a primitive event $\text{AtomicProcessEndEvent}$ is detected as well — notice that in this case the particular ordering or the time when both primitive events were detected do not matter.

With event expressions introduced we can extend the notion of event occurrences to composite events.

**Definition 4** (Composite Event Occurrence). A composite event occurrence is a union of $n$ primitive event occurrences, where $n > 0$, i.e., a composite event occurrence $e = \{\langle E_i, e_i, t_i \rangle, \ldots, \langle E_n, e_n, t_n \rangle\}$, where $E_i$ is an event type, $e_i$ is a primitive event instance of $E_i$, and $t_i \in T$ is a time when event $e_i$ occurred, for $i = 1, \ldots, n$.

Since composite events are not instantaneous but stretch over an interval, the occurrence interval for a composite event occurrence needs to be defined. For an event occurrence $e$ we define the functions $\text{start}$ and $\text{end}$ as follows:
\[
\begin{align*}
\text{start}(e) &= \min\{t \mid \langle E, v, t \rangle \in e\} \\
\text{end}(e) &= \max\{t \mid \langle E, v, t \rangle \in e\}
\end{align*}
\]

Intuitively, for an event expression \( A \), a composite event occurrence is represented by those primitive event occurrences that caused the occurrence of the event described by the expression \( A \). We use the notion of \textit{event streams} to define the precise semantics of event expressions. Formally, occurrences of composite events are derived from the definition of event streams as well.

**Definition 5 (Event Streams).** A \textit{primitive event stream} is a set of primitive event occurrences of the same event type with different times. A \textit{general event stream} is a set of event occurrences. Event streams can be combined by means of composition functions. For event streams \( R \) and \( S \) and time \( t \in T \) we define the following composition functions:

\[
\begin{align*}
\text{dis}(R, S) &= R \cup S \\
\text{con}(R, S) &= \{r \cup s \mid r \in R \land s \in S\} \\
\text{neg}(R, S) &= \{r \mid r \in R \land \exists s \in S \land \text{start}(r) \leq \text{start}(s) \leq \text{end}(s) \leq \text{end}(r)\} \\
\text{seq}(R, S) &= \{r \cup s \mid r \in R \land s \in S \land \text{end}(r) \leq \text{start}(s)\} \\
\text{tim}(R, t) &= \{r \mid r \in R \land \text{end}(r) - \text{start}(r) \leq t\} \\
\text{delay}(R, t) &= \{q \mid r \in R \land q \text{ is a new temporal event refering to } r \land \text{start}(q) \leftarrow \text{end}(q) \leftarrow \text{end}(r) + t\}
\end{align*}
\]

The idea of defining the semantics of event expressions by means of event streams is to define an \textit{interpretation} of event expressions that maps event expressions to event streams. By doing so we get a specific realization of event expressions. An interpretation function maps event types to corresponding primitive event streams and it uses the introduced composition functions to map complex event expressions to general event streams.

**Definition 6 (Semantics of Event Expressions).** The semantics of event expressions is defined by means of an \textit{interpretation} \( I \) which is a function that maps each event type \( A \in \mathcal{P} \) to a primitive event stream containing primitive event occurrences of the type \( A \). We use the notation \([A]^I\) for an interpretation of \( A \). An interpretation of event expressions is defined as follows:
\[
\begin{align*}
\lbrack A \rbrack^T &= \mathcal{I}(A) \text{ for } A \in \mathcal{P} \\
\lbrack A \lor B \rbrack^T &= \text{dis}(\lbrack A \rbrack^T, \lbrack B \rbrack^T) \\
\lbrack A \land B \rbrack^T &= \text{con}(\lbrack A \rbrack^T, \lbrack B \rbrack^T) \\
\lbrack A - B \rbrack^T &= \text{neg}(\lbrack A \rbrack^T, \lbrack B \rbrack^T) \\
\lbrack A; B \rbrack^T &= \text{seq}(\lbrack A \rbrack^T, \lbrack B \rbrack^T) \\
\lbrack A \rbrack^T &= \text{tim}(\lbrack A \rbrack^T, t) \\
\lbrack A + t \rbrack^T &= \text{delay}(\lbrack A \rbrack^T, t)
\end{align*}
\]

Such definition of the event expressions semantics results in an algebra with a simple semantics and intuitive algebraic properties. For example, Carlson and Lisper (2004) show the commutativity of $\land$, $\lor$ operators, associativity of $\land$, $\lor$, $;$ and distributivity of $\land$, $\lor$. According to the definition of event expression semantics, given an interpretation function $\mathcal{I}$, each event expression identifies an event stream with all composite event occurrences that match the given expression. This makes an event algebra and event expressions a useful tool for detection of composite events as we discuss later in this chapter.

**SEMANTIC FILTERING**

So far we discussed event types of primitive events and the structure of composite events that is defined by means of event algebra expressions. We did not speak about the content of events and about means of detecting events depending on their content, which is the goal of this section. In this section, we extend the event detection algebra by introducing event filtering based on expressions in the form of conjunction of description logic atoms enriched with OWL datatypes. The format of filtering expressions corresponds to conjunctive queries. Such expressions allow us to match events represented as OWL instances.

**Definition 7** (Filter Expression). 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), $\text{sameAs}(s,t)$ (same as atom), $\text{differentFrom}(s,t)$ (different from atom), and $\text{builtinID}(d_1,\ldots,d_n)$ (built-in atom), where $C$ is an OWL class name, $Po$ is an OWL object property, $Pd$ is an OWL datatype property, $s$ and $t$ are variables or OWL individuals, and $d$ is a variable or an OWL data value.

For evaluation of filter expressions we assume existence of an OWL knowledge base $KB = \langle T, A \rangle$, with $T$ being a TBox, i.e., a terminology, and $A$ being an ABox, i.e., an instance part of the knowledge base. An execution engine needs to maintain the $KB$ during execution of the process model and stores produced results in the $KB$.

A filter expression $f$ holds with respect to $KB$, if there exists an assignment of individuals and data values from $KB$ to all free variables in the expression, such that all its atoms hold, i.e.,
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 $builtinID(d_1,\ldots,d_n)$ holds if the built-in relation $builtinID$ holds on the interpretations of the arguments (see Horrocks et al. 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.

**Definition 8** (Restricted Event Type). A restricted event type $T$ is defined as

$$ T = \forall v : A[f] $$

where $T$ is the name of the defined restricted event type, $A \in \mathcal{P}$ is an event type, $\forall v$ is a variable which is used to refer to a primitive event occurrence that is detected as an instance of $A$, and $f$ is a filter expression.

Notice that restricted event types provide a mechanism for definition of new, domain or application specific event types. This way our formalism supports specification of concepts of process specific monitoring ontologies (as introduced in Figure 2). 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 for this event occurrence, the event occurrence is detected also as an instance of $T$. A variable $\forall v$ can be used in the expression $f$ to refer to the detected event occurrence of $A$.

**Example 3.** The following expression defines a new restricted event type $Logout$ derived from the $AtomicProcessEndEvent$ event type:

$$ Logout = \forall x : AtomicProcessEndEvent[
process(\forall x,"shoppingService;Logout")]] $$

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. The $process$ property in this example is the property defined for every event instance in the events ontology (see Figures 3 and 5).

We introduce restricted event expressions by allowing restricted event types in place of ordinary event types.

**Definition 9** (Restricted Event Expression). A restricted event expression is an event expression
in which defined restricted event types can be used in the same fashion as event types.

The following examples make use of process definitions introduced in Example 1 (as detailed in Table 1 and in Figure 1). In the following examples we assume that restricted event types Login, LookupItem, AddItemToCart and BuyAndShipItems were defined in a similar fashion as the Logout restricted event type in Example 3.

**Example 4.** The following restricted event expression can be used to detect situations in which a customer logs out without buying anything (event pattern 2b from Example 1):

\(((\text{Login};\text{Logout}) - \text{BuyAndShipItems})\)

Notice 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. This is a very important property of restricted event expressions since it will allow us to employ the efficient detection techniques of event detection algebras to restricted event expressions. As an outcome restricted event expressions are suitable for runtime event detection. At the same time, the expressive power of restricted event expressions is somewhat constrained because the filter expressions can be applied only to event types. To address this limitation, we define *extended event expressions* as event expressions in which a filter expression can be attached to any event subexpression.

**Definition 10 (Extended Event Expression).** If \( A \) is a valid event expression, then also \( A[f] \), \(?v:A\) and \(?v:A[f]\) are valid event expressions, where \( f \) is a filter expression and \(?v\) is a variable identifying some event occurrence from the event stream of \( A \) which was detected as an instance of \( A \). In extended event expressions, restricted event types can be used as well.

Extended event expressions offer more expressive power than restricted expressions, since filters attached to any subexpression can access and compare several event occurrences. There is a price in terms of a worse efficiency during detection of composite events, as we show in a bit. The interpretation of extended event expressions is again straightforward. During event detection of an expression \(?v:A[f]\), for every event occurrence in the stream of expression \( A \) the filter expression \( f \) is evaluated, and only if it holds for this event occurrence, the event occurrence is added into the stream of the extended event expression. Before \( f \) is evaluated, all variables in \( f \) that are referring to some subexpression of \( A \) are substituted by corresponding event occurrences.

**Example 5.** The following extended event expression detects 2 Login calls of the same user (i.e., the value of the "&shoppingService;username" input is the same) within an interval of 120 seconds. It is a variation of the problem 2a in Example 1:
Example Event Expressions with Semantic Filtering

To demonstrate the power of event expressions with semantic filtering, we present some more examples. Where possible, we use a restricted event expression instead of an extended event expression.

Example 6. The following restricted event type detects \textit{LookupItem} service calls with the \textit{category} parameter value that is an instance of the \textit{Book} class:

\[
\text{LookupItemBook} = \exists x : \text{AtomicProcessStartEvent}[ \\
\quad \text{process}(x,"\text{shoppingService;}\text{LookupItem}") \land \\
\quad \text{input}(x,? \text{param}) \land \\
\quad \text{toParameter}(? \text{param},"\text{shoppingService;}\text{category}") \land \\
\quad \text{objectValue}(? \text{param},? \text{value}) \land \\
\quad \text{Book}(? \text{value})]
\]

Example 7. The following restricted event expression detects a situation where 3 \textit{Login} calls were executed within an interval of 120 seconds:

\[
(\text{Login};\text{Login};\text{Login})_{\text{120}}
\]

Example 8. The following restricted event expression detects a situation where 3 unsuccessful \textit{Login} calls were executed within an interval of 120 seconds:

\[
(\text{FailedLogin};\text{FailedLogin};\text{FailedLogin})_{\text{120}}
\]

The \textit{FailedLogin} restricted event type is defined as follows:
FailedLogin = \( ?x : \text{AtomicProcessEvent}\) [  
\[\text{process}(?x, "shoppingService; Login") \land \text{output}(?x, ?param) \land \text{toParameter}(? \text{param}, "shoppingService; status") \land \text{dataValue}(? \text{param}, "false")\]  

**Example 9.** The following extended event expression detects service calls taking longer than 60 seconds:

\( ((?\text{start} : \text{AtomicProcessStartEvent}; (?\text{delay} : \text{AtomicProcessStartEvent} + 60)) - \text{LoginEnd})\)

\( ?\text{end} : \text{AtomicProcessEvent}\) [  
\[\text{process}(?\text{start}, ?\text{processInstance}) \land \text{process}(?\text{delay}, ?\text{processInstance}) \land \text{process}(?\text{end}, ?\text{processInstance})\]  

The same problem as the one in Example 9 can be solved by using a restricted event expression if the service of interest is known in advance.

**Example 10.** The following restricted event expression detects service calls of the *Login* service taking longer than 60 seconds:

\( ((\text{LoginStart}; (\text{LoginStart} + 60)) - \text{LoginEnd})\)

The *LoginStart* and *LoginEnd* restricted event types are defined as follows:

\[\text{LoginStart} = ?x : \text{AtomicProcessStartEvent}\] [  
\[\text{process}(?x, "shoppingService; Login")\]  

\[\text{LoginEnd} = ?x : \text{AtomicProcessEvent}\] [  
\[\text{process}(?x, "shoppingService; Login")\]  

**EVENT DETECTION AND PROCESSING**

During execution, primitive event occurrences are emitted and/or detected by the system. The main problem of event detection is to identify event occurrences that form event streams of given event expressions. In the case of an expression that has a simple form of an event type or a restricted event type, we need to decide which emitted primitive event occurrences belong into the stream of the particular (restricted) event type. In the case of complex event expressions, which describe composite events, the situation is somewhat more complicated. An occurrence of a composite event is caused by occurrences of primitive events that match the given event expression. This means that in order to detect event occurrences matching a specified composite
event expression, such combinations of primitive event occurrences need to be identified that form the matching composite events. We start with considering detection of primitive events (i.e., those matching event types and restricted event types), and next we consider techniques for detection of composite events.

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 $e = \langle \{E,v,t\} \rangle$ be an event occurrence of an event type $E$. Naturally, since $e$ is an instance of type $E$, it also is an instance of all superclasses of $E$. This means, that if we are given an event type $T$, the only thing that needs to be done to check whether $e$ belongs into the event stream of $T$ is to test if $T$ is a superclass of $E$. This can be done very efficiently since the set of superclasses can be precomputed for each event type.

Next, if $T$ is a restricted event type, i.e., $T = ?x:A[f]$ with $f$ being a filter expression, additionally $f$ needs to be evaluated. After a primitive event occurrence $e$ is detected that belongs into the event stream of an event type $A$, the filter expression $f$ must hold for $e$ assigned to $?x$ in order to $e$ be added to the primitive event stream of $T$. We mentioned that filter expressions correspond to conjunctive queries and therefore evaluation of filter expressions can be based on answering conjunctive queries with respect to the knowledge base $KB$. Since during runtime detection the $KB$ contains only instances produced within this session, the size of $KB$ is relatively small and the query can be evaluated efficiently. One way of evaluating the filter expression is to translate it into a SPARQL query (Prud’hommeaux and Seaborne, 2008) and to use existing tools to answer the query.

**Composite Events Detection**

In order to detect composite event occurrences we adopted an approach based on event detection trees proposed by Chakravarthy et al. (1994) and we modified the algorithm according to the work of Carlson and Lisper (2004) which presents a more efficient technique suitable especially for runtime detection. We extended the algorithm with evaluation of 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. Consider for example an event expression $(\text{LookupItem}; \text{AddItemToCart})$. This expression represents composite events in which an event instance of $\text{LookupItem}$ occurs before an event instance of $\text{AddItemToCart}$. Assume the event stream of the $\text{LookupItem}$ subexpression contains several matching primitive event occurrences. If we are interested in all occurrences matching $(\text{LookupItem}; \text{AddItemToCart})$ expression, then whenever a new instance of $\text{AddItemToCart}$ type is detected, a composite event occurrence needs to be detected for all pairs of this new $\text{AddItemToCart}$ event occurrences and the instances in the $\text{LookupItem}$ stream. Generally, due to the combinatorial explosion many occurrences of a composite event need to be
detected, especially if primitive events can occur repeatedly. In most situations, such an approach is not desirable and even not realistic because of time and memory constraints.

Chakravarthy et al. (1994) defined several policies that restrict the amount of detected composite event occurrences to deal with the combinatorial explosion. We will focus on the recent policy which is suitable especially for runtime event detection. The recent policy maintains an invariant that at each point in time at most one event occurrence for a given event expression is detected. If there are more candidate occurrences, the one with the maximal start time is detected, i.e., the most recent occurrence. Formally, the recent policy can be defined by specifying conditions that need to hold for an event stream $S'$ which we get by applying the recent policy to an event stream $S$.

**Definition 11** (Recent Policy). Let $S$ and $S'$ be event streams. An event stream $S'$ adheres to the recent policy with respect to the stream $S$ if the following conditions hold:

1. $S' \subseteq S$
2. $\forall e \in S \ \exists e' \in S' \ (\text{start}(e) \leq \text{start}(e'))$
3. $\forall e_1, e_2 \ ((e_1 \in S' \land e_2 \in S' \land \text{end}(e_1) = \text{end}(e_2)) \Rightarrow e_1 = e_2)$

The recent policy guarantees that if there exist one or more event occurrences of a composite event at some point in time, one of them will be always detected in the event stream adhering to the recent policy. This means we will not miss any important event occurrence. This is an important property required in many applications contexts. The only thing that can be missed are possible duplicate event occurrences that occur at the same time.

Carlson and Lisper (2004) show 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 techniques adhering to the recent policy can be implemented very efficiently with respect to time and memory requirements. There are two reasons for that: (1) since at every point in time only one event occurrence (the most recent one) of a given event expression needs to be detected, there is no need for generating all possible combinations and thus the combinatorial explosion can be avoided, and (2) for the same reason only the most recent primitive event occurrence of each primitive event type needs to be remembered, instead of remembering all occurrences. This makes detection algorithms based on the recent policy suitable for runtime detection of composite event occurrences.

In the previous section, we observed that allowing restricted event types (i.e., event types enriched with semantic filters) in event expressions does not have any effect on properties of the underlying event algebra. As an outcome, 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. The reason is that, when the recent policy detection techniques are applied to extended expressions, some composite event occurrences can be missed, which breaks the Condition 2 of Definition 11. 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 event occurrences can be missed.

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

**Event Detection Trees**

Event detection trees (Chakravarthy et al., 1994) present an efficient mechanism for detection of composite events defined by event expressions. For detection purposes, an event expression is represented as a tree. Each event type occurring in the expression is represented as a leaf, and every composition operator in the expression is an interior node in the tree. The tree represents a decomposition of the event expression into its subexpressions: the root stands for the whole expression, each interior node in the tree represents a subexpression, and leaves represent event types. In terms of the structure, an event detection tree can be seen as a parsing tree which represents the syntactic structure of the expression.

Detection of composite event occurrences by event detection trees proceeds as follows:

1. detection starts at the bottom in leaves with detecting primitive event occurrences:
   - for a primitive event occurrence $e = \langle E, v, t \rangle$ each leaf node representing an event type $T$ is notified if $T$ is a superclass of $E$

2. detection proceeds in the bottom up direction by progressively detecting occurrences of more complex subexpressions:
   - if a node $n$ representing an event expression $A$ is notified by some of its children $c$ about an occurrence of a new event $e$ detected by $c$, it tests if the newly detected event occurrence $e$ reported by $c$ causes an occurrence of a more complex event matching the subexpression $A$ represented by this node $n$
   - whenever a new event occurrence $e$ is detected by a node $n$, the node $n$ notifies its parent(s) about the newly detected event occurrence $e$
   - every node maintains a history of event occurrences in its own buffer; when the tree implements the recent policy, only the last event occurrence needs to be remembered in each node (with the only exception for the ; operator — see (Carlson and Lisper, 2004) for details)

3. detection ends when eventually the root is reached and an event occurrence for the whole expression is detected

We extended the original concept of detection trees by adding support for restricted types detection and for semantic filters. When an event occurrence of the primitive event type is detected, the node notifies all its parents to which it is connected. If the parent node represents a restricted event type, it evaluates the associated *filter expression* to test whether the reported
event occurrence matches this filter.

The internal processing of operator nodes implementing semantics of each event algebra operator is quite intuitive. For example, a node implementing the $\wedge$ (and) operator simply checks if an even occurrence was detected for both of its subexpressions. When the node is notified about a new event occurrence by one of its subexpressions, it fires only if it was already notified by the other subexpression before (i.e., both subexpressions need to notify the $\wedge$ node irrespective of the order so that $\wedge$ can fire). For technical details on implementation of operator nodes adhering to the recent policy we refer the reader to (Carlson and Lisper, 2004).

Described processing mechanisms are used during runtime detection of restricted event expressions. To support detection of extended expressions each operator node additionally evaluates the filter expression of the corresponding subexpression to decide whether a candidate event instance matches the filter or not.

**Implementation and Evaluation**

As a model for event processing we adopted the event-based model (Luckham, 2002; Muhl et al., 2006). We implemented described mechanisms for runtime detection of both primitive and composite events. Detection of primitive events uses the Jena library and the Pellet reasoner. Filter expressions are evaluated by translating them into SPARQL queries. Detection of composite events uses described event detection trees with the recent policy applied for runtime detection.

We performed a basic experimental evaluation of implemented algorithms by executing a real process model consisting of Semantic Web services. The process consisted of a sequence of three atomic processes (i.e., 3 service calls). The workflow was executed over the network. We were interested in the overhead of the monitoring infrastructure. We compared the execution times in 3 cases: (1) execution with monitoring infrastructure disabled, (2) execution with detection of primitive events only, and (3) detection with both primitive and composite events enabled. In cases 2 and 3 we registered several event expressions to be detected by the execution engine. We executed the workflow 200 times. On average, the execution time in case 1, i.e., with detection disabled, was 3.47 seconds (this includes rather complicated XML and XSLT transformations, preconditions evaluation and knowledge base updates). In case that monitoring of primitive events was enabled (case 2), the execution engine generated on average 95 primitive events during the whole workflow execution and the overall execution time involving detection of primitive event expressions was on average by 3.59ms (0.1033%) worse than in case 1. When composite event detection was enabled (case 3) the overall execution time involving detection was on average by 14.7ms (0.4232%) worse than in case 1. Although in real scenarios performance will depend on many factors such as network performance, specific detection patterns, or complexity of monitored workflow, it seems to be clear that in context of distributed process models execution using web services, the overhead of semantic monitoring is rather negligible and that it can be employed without major concerns.

**CONCLUSIONS**
In this chapter we described the semantic monitoring model consisting of a modular monitoring ontology and efficient event detection techniques. Specifically, we defined a taxonomy of generic event types, and two modules defining event type taxonomies specific to OWL-S semantic web services and to the business artifact-centric approach. The proposed monitoring approach combines semantically rich primitive events with an event algebra used for specification of composite event patterns. We augmented the event algebra with semantic filtering that allows detection of events with semantically rich content. Furthermore, we identified a restricted combination of the event algebra and semantic filters that is suitable for runtime monitoring, while we proposed an unrestricted variant to be used for offline analysis since it allows more complex event patterns to be detected. We developed algorithms for detection of primitive and composite semantic events that can be used with both the restricted and the unrestricted version of the event algebra.

The semantic monitoring approach allows to overcome various heterogeneities (lexical, syntactic, schema, etc.) by providing an abstraction layer using shared vocabulary in the form of shared ontologies. Such ontologies can either be existing domain ontologies or can be created as an outcome of business modeling, e.g., based on approaches such as business artifact-centric modeling. Among other main advantages of the semantic monitoring model are its application independence, flexibility and extendability. 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 on the monitoring tools developers.

Currently we are exploring the possibilities of employing techniques of semantic monitoring to support business intelligence and business analytics. Specifically, we focus on employing a semantic overlay over business process execution histories of legacy systems in order to enable better visibility and insights into operations of process executions, and possibly allow their optimization and re-factoring. Another topic for future work is to address the problem of simplifying or automating creation of the mappings between the semantic (ontology) layer and legacy systems and real enterprise systems. This problem is hard and probably cannot be fully automated, but it can be substantially simplified by providing the users with suitable tools that support creation of the mappings.

REFERENCES


Recommendation, http://www.w3.org/TR/owl-ref/.


Management. Springer.


Notice that since composite event occurrences consist of several primitive events which occurred within some time interval, it is meaningful to think about composite events as phenomena with a well defined duration, as opposed to being completely instantaneous as primitive events. Interestingly, we clearly distinguish composite events from concepts such as tasks, activities or processes. Specifically, we consider composite events as indicative phenomena which only reflect that something has happened in the observed system – e.g., a process was executed (the actual phenomenon).

In essence, the main advantage of recent policy based techniques, which is the fact that only the most recent event occurrences need to be detected / generated and remembered, turns out to be the problem for detection of extended event expressions. To demonstrate this, let us consider that we attach a filter $f$ to the event expression $E = (\text{LookUpItem;} \text{AddItemToCart})$ by which we get an extended expression $E' = ?x:(\text{LookUpItem;} \text{AddItemToCart})[f]$ . Now, let us consider the situation where at a particular time (say time point 3) there are two candidate composite event occurrences (one starting at time 1 and the other one at time point 2) that can be detected before applying the filter $f$ . Let us assume that the expression $f$ holds only for the event occurrence which starts at point 1 (we label it $e_1$ ) and that $f$ does not hold for the composite event occurrence which starts at point 2 (let us label it $e_2$ ). Since detection techniques using the recent policy will only consider the event occurrence $e_2$ (i.e., the one with the later start) for which $f$ does not hold, no composite event will be detected for expression $E'$ at time 3. However, when no policy is applied, a composite event $e_1$ is detected, which leads to a contradiction with property 2 of the recent policy definition 11.