Moving Active Functionality from Centralized to Open Distributed Heterogeneous Environments

M. Cilia *, C. Bornhövd **, A. Buchmann

Databases and Distributed Systems Group, Department of Computer Science
Darmstadt University of Technology - Darmstadt, Germany
{cilia,bornhoev,buchmann}@informatik.tu-darmstadt.de

Abstract. Active functionality is especially useful for enforcing business rules in applications, such as Enterprise Application Integration (EAI) and e-commerce. It can be used as glue among existing applications, and for data transformations between heterogeneous applications. However, traditional active mechanisms have been designed for centralized systems and are monolithic, thus making it difficult to extend and adapt them to the requirements imposed by distributed, heterogeneous environments. To correct this we present a flexible, extensible, service-based architecture built on ontologies, services and events/notifications. The main contributions of this work are: i) the homogeneous use of ontologies for a semantically meaningful exchange and combination of events in open heterogeneous environments, and for the infrastructure itself; ii) a flexible architecture for the composition of autonomous, elementary services to provide Event-Condition-Action (ECA) functionality in different configurations; iii) the interaction of these services via notifications using a publish/subscribe mechanism (concept-based addressing).

1 Introduction

Active database systems (aDBMS) enhance traditional database functionality with powerful rule processing capabilities. The database system performs certain operations automatically in response to events occurring and certain conditions being satisfied. Active functionality is especially useful for enforcing business rules. However, traditional active mechanisms have been designed for centralized systems and are monolithic, thus making it difficult to extend or adapt them. New large-scale applications, such as EAI, e-commerce or Intranet applications impose new requirements. In these applications, integration of different subsystems and collaboration with partners' applications is of particular interest, since business rules are out of the scope of a single application. For example, consider the following business rule: "when the volume of corn falls below 1200 tons, and the corn comes from Euroland, and Euro/US$ is under 0.8 then bid for all at 2.30 US$ per ton, and also notify the manager". Involved events and data are coming from diverse sources, here and also the execution of actions is performed on different (sub-) systems. Observe that events and actions are not necessarily directly related with database operations.

In such scenarios, the required active functionality should be moved outside of the active database system by offering a flexible service that runs decoupled from the

* Also ISISTAN, Faculty of Sciences, UNICEN, Tandil, Argentina
** Current affiliation: Hewlett-Packard, e-Solutions Divison, CA, Cupertino

database, and that can be combined in many different ways and used in a variety of environments. A service-based architecture seems to be appropriate, in which an active functionality (ECA rule) service can be seen as a composition of other services, like event detection, event composition, condition evaluation, and action execution. Thus, services can be combined and configured according to the required functionality, as proposed by the unbundling approach in the context of aDBMSs [11, 12, 18]. Whether the unbundling approach is realistic for database systems or not, it is inadequate for distributed environments since DBMS components to be ”unbundled” are designed with homogeneous, centralized environment in mind. Combining components developed by different, independent providers leads inevitably to problems if the meaning of terms employed by different components and data to be exchanged is not shared. A similar problem is encountered when integrating heterogeneous applications.

In addition to the difficulties introduced by heterogeneity, the inherent characteristics imposed by large-scale distributed environments, in particular, impact the following issues of active functionality: event exchange mechanism, event semantics (correct interpretation and use of events), event filtering, and complex event detection.

In this paper we present a service-based architecture that supports the use of active functionality in various environments. In this context we focus on aspects related with the problems of interpreting event contents coming from different systems and sources. Despite the fact that event composition is an important issue we do not present here details about complex event detection since they are out of the scope of this paper.

The rest of this paper is organized as follows. Section 2 provides a discussion of the most important aspects and proposals for moving from centralized active functionality to distributed heterogeneous environments. In Section 3 the conceptual foundation of our proposal for a flexible and extensible active service for this kind of environment is presented. In Section 4 our reference architecture is introduced. Finally, in Section 5 we present conclusions, address open issues, and discuss future work. Examples throughout the paper are related to online auctions.

2 Moving to Open Distributed Heterogeneous Environments

Active database functionality is developed for a particular DBMS, becoming part of a large monolithic piece of software (the DBMS itself). Monolithic software is difficult to extend and adapt. Moreover, active functionality tightly coupled to a concrete database system hinders its adaptation to today’s Internet applications, such as, e-commerce, where heterogeneity and distribution play a significant role but are not directly supported by traditional (active) databases [18].

Another weakness of tightly coupled aDBMSs is that active functionality can not be used on its own, without the full data management functionality. However, active functionality is also needed in applications that require no database functionality at all, or only simple persistence support. As a consequence, active functionality should be offered not only as part of the DBMS, but also as a separate service which can be combined with other services to support Internet-scale applications. Implications of heterogeneity, distribution and active functionality as an independent service, and a review of other approaches that address these issues are discussed in the following subsections.

2.1 Heterogeneity

The integration of events coming from heterogeneous sources is comparable to the problems faced in federated multi-database systems. Similar to heterogeneity in data, we also have heterogeneity in events (which can be understood as containing event-descriptive data) coming from different sources. In C^2offein [19], event sources are encapsulated by means of wrappers. These wrappers map application-specific events into a shared event description composing syntax and structure. [18] propose abstract connectors to hide a set of heterogeneous components, making invisible the fact that different event sources exist.

To integrate data/events from different sources, the approaches mentioned above concentrate on structural aspects, and assume global knowledge by the DBA or the application developer of the assumed semantics of all relevant data and events. This assumption is unrealistic in a large and very dynamic environment like the Internet. Notice that data/events must be compared or correlated externally from the source generating the event. Events containing date or price attributes, require explicit knowledge about modeling assumptions in regard to format or currency to correctly interpret them. Moreover, consumers and producers of events may be previously unknown, therefore a common structural and semantic representation basis (common vocabulary) of the involved events is necessary for their correct interpretation and use [3]. Consider an online auction scenario, where participants may have never met before. Here a common vocabulary is mandatory (normally established using categories of items) and because of its global scope, representations of all the descriptive information require context information, such as, date and time format, metric system, currency, etc. to correctly interpret data.

2.2 Distribution: Detection of global composite events

There are several approaches that deal with distribution, in the area of event propagation [6, 14, 8, 15]. However, they do not consider event composition, where order between events is required to apply event operators (e.g. sequence), or to consume events coming from different locations. Normally, events are timestamped to provide a time-based order, but in open distributed environments global time is not applicable. In [17, 25] a global time approximation, known as *2g-precedence*, is proposed. Schwiderski [24] adopted the 2g-precedence model to deal with distributed event ordering and composite event detection. She proposed a distributed event detector based on a global event tree and introduced 2g-precedence based sequence and concurrency operators. However, event consumption is non-deterministic in the case of concurrent or unrelated events. Additionally, the violation of the granularity condition (2g) may lead to the detection of spurious events.

Many projects on event composition in distributed environments, such as [2, 22, 13, 27], either do not consider the possibility of partial event ordering or are based on the 2g-precedence model. Therefore, they suffer from one or more of the following drawbacks: they do not scale to open systems, they provide the possibility of spurious events, and present ambiguous event consumption. In [20] a new approach for timestamping events in large-scale, loosely coupled distributed systems is proposed that uses accuracy intervals with reliable error bounds for timestamping events that reflect the inherent inaccuracy in time measurements .

2.3 Unbundling of active database functionality into reusable components

Unbundling is the activity of decomposing systems into a set of reusable components and their relationships [12]. Unbundling active databases consists in separating the active part from active DBMSs and breaking it up into units providing services like, event detection, rule definition, rule management, and execution of ECA rules on one hand and persistence, transaction management and query processing services on the other [11]. A separation of active and conventional database functionality would allow the use of active capabilities depending on given application needs without the overhead of components not needed. Similar approaches were adopted by [7, 10, 19].

From our point of view, unbundling active functionality from a concrete system and then rebundling the corresponding components to apply them in an open distributed environment is not feasible. Unbundling in this context means to give up the ”closed world” which traditionally underlies a DBMS. Inherent characteristics of open distributed environments impose new requirements that were not considered in centralized environments, like the partial order of events. The consideration of these characteristics has direct impact on the event detector, which is the essential component of an aDBMS [5]. Consequently, it would not be feasible to reuse components taken from centralized DBMSs since they ignore relevant aspects of the new scenario. In addition, the semantics of operators and consumption modes are hard-wired in the code of existing event detectors. Because in the projects mentioned above a generic architecture is defined, there may be difficulties when integrating unbundled and newly developed (autonomous) components. Notice that the meaning of terms employed by different components can conduct to misinterpretations if a common semantic basis, i.e., vocabulary, is not shared.

2.4 Processing ECA-Rules

Rule execution semantics prescribe how an active system behaves once a set of rules has been defined. Rule execution behavior can be quite complex [26, 5], but we restrict ourselves to discussing essential aspects. The whole ECA-rule processing process is a sequence of four steps:

1. Complex event detection: event instances generated at event sources feed the complex event detector, which selects and consumes these events to detect specified situations of interest. It binds related event instances with the signaled event.
2. Rule selection: find fireable rules, and apply a conflict resolution policy if needed. There are three basic strategies: a) one rule is selected from the fireable pool, after rule execution the set is determined again, b) sequential execution of all rules in an evaluation cycle, and c) parallel execution of all rules in an evaluation cycle.
3. Condition evaluation: selected rules receive the signaled event instance as a parameter to allow the condition evaluation code to access event information (binding). For some applications it may be useful to delay the evaluation of a triggered rule’s condition or the execution of its action until the end of the transaction, or execute them in a separate transaction. These possibilities yield the notion of *coupling modes* [9].
4. Action execution: if the corresponding condition is satisfied context information (signaled event instance) is passed as a parameter to allow the action code to access

event information (binding). Transaction dependencies between the evaluation of a rule's condition and the execution of a rule's action are specified using Condition-Action coupling modes.

3 ECA architecture for distributed, heterogeneous environments

The goal is to provide ECA-rule processing functionality with characteristics similar to a centralized aDBMSs in a distributed component system to support new generation of large scale applications. The active functionality service proposed here is based on a flexible architecture founded on autonomous, combinable and possibly distributed services. Here ontologies play a fundamental role; we use them homogeneously to deal with the integration of events and the interaction of autonomous services. In addition, because our architecture is based on components, ontologies are fundamental for the interaction among components developed independently. The underlying communication between these services is based on a publish/subscribe mechanism, which is suitable for distributed environments and offers other advantages as shown later in this section. In particular, this work puts emphasis on:

- a flexible architecture that can be adapted for different application scenarios,
- the use of an ontology to allow the integration of events coming from heterogeneous sources and also for the infrastructure itself,
- a platform for composition of events coming from heterogeneous sources in distributed environments that deals with partial orderings and the lack of a central clock, and
- providing an active service as composition of other elementary services.

Three main pillars are the basis of our work: an ontology-based infrastructure, event notifications, and a service-based architecture. In the next subsections these three aspects are presented; on this foundation we present our architecture in Section 4.

3.1 Ontology-based Infrastructure

In our context active functionality mechanisms are fed with events coming from heterogeneous sources. These events encapsulate data, which can only be properly interpreted when sufficient context information about its intended meaning is known. In general this information is left implicit and as a consequence it is lost when data/events are exchanged across institutional or system boundaries. For this reason, to exchange and process events from independent participants in a semantically meaningful way explicit information about its semantics in the form of additional metadata is required.

Our architecture is based on the concepts developed in [4] where shared concepts (ontologies) are expressed through common vocabularies as a basis for interpretation of data and metadata. We represent events, or event content to be precise, using a self-describing data model, called MIX [3]. In the following we refer to events represented based on MIX, i.e. based on concepts from the common ontology, as *semantic events*. MIX refers to concepts from a domain-specific ontology to enable semantically correct interpretation of events, and supports an explicit description of the underlying interpretation context. Simple attributes of an event are represented as triples of the form $SSO = < C,\ v,\ \$ >$, with C referring to a concept from the common ontology, v representing the actual data value, and $\$$ providing additional metadata (represented

also as MIX objects) to make implicit modeling assumptions explicit. For instance, $SSO = < BidAmount,\ 99,\ \{< Currency, "USD" >\} >$.

Complex objects are represented in the form $CSO = < C,\ \mathbb{A} >$, with C referring to a concept of the ontology, and $\mathbb{A}$ providing the set of simple or complex objects representing its sub-objects. For example, a PlaceBid event can be represented as:

CSO = <PlaceBid, {<ParticipantId,412>,
<ItemId,5423>,
<BidAmount, 99,{<Currency,"USD">}>,...}>

Semantic events from different sources can be integrated by converting them to a common semantic context using conversion functions defined in the ontology [3]. In our work ontologies are used at three different levels: a) the basic level, where elementary ontology functionality and physical representation is defined; b) the infrastructure level, where concepts of the active functionality domain are specified; and c) the domain-specific level, where concepts of the subject domain (e.g. online auctions) are defined.

Basic representation ontology: Here elementary ontology functionality and physical representation concepts like strings, booleans, numbers, lists, etc. are defined.

Infrastructure-specific ontology: All elements related with active functionality are represented with concepts defined here. Difficulties associated with different rule language dialects, ambiguities and imprecise terms are resolved using an explicit common vocabulary. For instance, issues related with the definition of rules like event, condition, action, and so on, are explicitly defined in our infrastructure-specific ontology.

Domain-specific ontology: With the purpose of their integration in mind, events are represented with concepts of a common vocabulary and with context information. Required real world concepts are defined in the domain-specific ontology. Based on this, conversion functions can be applied to integrate data/events and filters and complex detectors can be defined based on the common ontology without considering peculiarities of event representations and interpretation contexts coming from different sources.

This approach provides the following benefits:

- Independence of the rule definition language: Because there is an explicit definition of terms rule compilers can translate from ”any” rule specification to the known target vocabulary. Moreover, rules can be defined directly by applications using an API (accessing the ontology) or by the user in a user-friendly rule language definition.
- Extensibility: New aspects can be incorporated to the rule definition and to the service itself due to the extensibility provided by the underlying ontology support.
- Service interaction ”independence”: Service interfaces are defined using ontology-based concepts contributing to their clear understanding by service-clients and service-providers.

Furthermore, other aspects related with the infrastructure, in particular, notification-related terms (notification, operational data, detection-time, event source, priority, time-to-live, etc.) are also captured in the infrastructure-specific ontology.

3.2 Events and Notifications

An event is understood here as a happening of interest. Events can be classified as:

- Database events that are further subdivided into data modification and data retrieval events.
- Transaction events refer to the different stages of transaction execution, e.g. begin transaction, commit, rollback, etc.
- Temporal events are classified in absolute, periodic and relative. Absolute temporal events are defined using a particular day and time, while periodic temporal events are signaled repeatedly using time or calendar functions, e.g. every Friday at 11:59PM. Relative ones are defined using a time period with respect to another event, e.g. one week after BeginOfAuction.
- Abstract events or application-defined events are declared by an application, e.g. user-login, auction-canceled. Events of this kind are signaled explicitly by the application.

Observation of happenings include the use of interception and polling mechanisms. In distributed platforms, like CORBA and J2EE, service requests can be intercepted. Using this feature, happenings related with a method execution can be intercepted transparently (without modifying the application). On the other hand, there are event sources that may need to be polled in order to detect events.

The event classification presented above is explicitly specified in the infrastructure-specific ontology. Notice that ontologies in our architecture are extensible. For instance, the representation of a heartbeat[1] is based on the definition of the periodic temporal event, adding in this case some extra information, such as frequency, process identification, etc. Likewise, real-world aspects of a particular domain are represented at the domain-specific layer, e.g. BeginOfAuction as a specialization of an absolute temporal event; ParticipantLogin as application-defined, and so on.

A notification is a message reporting an event to interested consumers. A notification carries not only an event instance but also important operational data, such as reception time, detection time, event source, time to live, priority, etc. As seen on several active system prototypes, complex event detection is mainly based on operational data (particularly comparing timestamps of event instances) while filtering is based on both (i.e. event source and/or attribute values). For this reason, we distinguish the content of a notification into operational data and the event data itself. Operational data concepts are also defined as part of the infrastructure level.

Events coming from different applications are integrated by event adapters. They convert source-specific events into events represented by ontology-based concepts enriched with semantic contexts, i.e. semantic events (see Section 4.2).

3.3 Service-based ECA-rule Processing

We realize the ECA-rule processing as a combination of its basic services, i.e. complex event detection, condition evaluation, and action execution service. Event, condition and action services are then combined using a notification service based on a publish/subscribe mechanism which transports event information among them. Subscribers (consumers) place a standing request for events by subscribing. On the other hand, a

[1] The heartbeat protocol is based on a message sent between machines at a regular interval with the purpose to monitor the availability of a resource.

publisher makes information available for its subscribers. Thus, event producers and consumers are decoupled in our architecture. Our ECA-rule service offers the functionality needed to define, remove, activate/deactivate, and search/browse ECA-rules. Figure 1 depicts a configuration of our elementary services using boxes to denote services and lolipops for their interfaces. Circles inside these boxes represent instances of objects under the control of the corresponding service.

Now consider the registration of a rule R1 with the ECA-rule service, as shown in step 1 in Figure 1. Following this is the registration of its corresponding event, condition and action with the proper services (step 2). The complex event detector configures internal objects in order to detect R1's event; then the condition evaluation service instantiates a condition object and subscribes it with R1's event object (step 3). Afterwards, the action execution service instantiates an action object and subscribes it with R1's condition object completing the subscription phase.

Once a triggering event is detected, the complex event detector publishes this happening. That means, all rules that were defined using this triggering event are automatically "activated", in particular their condition objects are notified (step 4a). In this situation, no conflict resolution policy is needed because all rules are executed concurrently (other execution models are possible). When condition objects are notified, they evaluate their predicate and if true, automatically notify the corresponding action objects using the same notification service (step 4b).

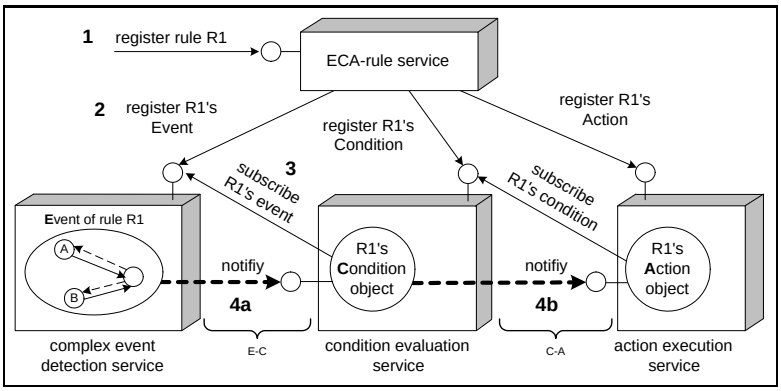

Fig. 1. Interaction among the elementary services (E-C-A)

Services communicate using a notification service. The notifications sent contain a representation of the triggering event, with information about the happening of interest and its context. The communication among these services could be done using other mechanisms, e.g. Remote Procedure Call. However, the publish/subscribe mechanism plays an important role in our architecture providing the following advantages:

- it allows asynchronous communication and decouples event producers and consumers (suitable for open distributed environments),
- it is particularly useful if various rules are associated with the same event,
- it facilitates concurrent rule execution (without centralized rule selection mechanism),
- the notification contains required event information and its context (context propagation),

– it provides a simple and powerful generic communication model,
– it supports location transparency due to subject organization, and
– it helps to explicitly represent dependencies of the flow of work.

Elementary services expose two kinds of generic and very simple interfaces:

– Service interface: defined as a single method that receives as a parameter an event notification which is represented based on the common ontology.
– Configuration interface: for administration purposes, such as register, activate, deactivate, delete, etc.

This simple service interface definition provides flexibility, allowing to configure the flow of service execution easily. For instance, consider the omission of the Condition part in a rule definition (EA-rules). This changes the flow of execution where the Action connects directly to the Event detection. Similarly, event filters can be placed between event source and the complex event detector without requiring to change the code of these components. The destination of the outgoing notifications relies on *concept-based addressing*, which is used for notification dissemination (see Section 4.1).

In addition to the elementary services mentioned above the complex event detection service combines other services which are required for this event detection, like, filtering service, time service, alarm service, event adapters. Furthermore, the following services are used as part of our architecture: ontology service, repository service, and notification service. All these services are described in more detail in the next section.

4 Reference Architecture

Our work is based on a service architecture for extensibility and flexibility reasons. Figure 2 shows, a combination of the basic components to support typical active functionality. Services are implemented using loosely-coupled components to fit distribution requirements. The communication among these services is based on a notification service using a publish/subscribe mechanism. Dashed arrows in Figure 2 depict the use of the underlying notification service.

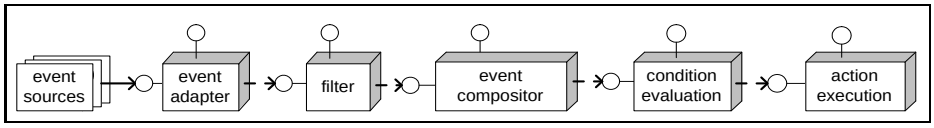

Fig. 2. Basic components to support active functionality

The services' simple interfaces provide flexibility in order to configure the service execution flow independently of the next service interface in the process chain. In our particular case, services are combined in a chain beginning with the event source and ending with the action execution service. In between optional services, such as event filters, and condition evaluation can be interconnected.

Information about rule definitions, deployment and configuration of their events, conditions and actions, and knowledge about services, is maintained in a repository to support configuration and maintenance. In addition, an ontology service is used to manage concept specifications defined at the three layers of our infrastructure.

4.1 Notification (Event delivery) Service

A publish/subscribe mechanism decouples producers and consumers of messages. Instead of addressing by location (i.e. by IP number or socket address), publish-subscribe interaction uses the generic communication paradigm of *subject-based addressing*. Producers of messages publish their individual messages under certain subjects. They do this without any knowledge of who or what applications are subscribing to these subjects. Subjects are used to direct messages to their destinations, so applications can communicate without knowing the details of network addresses or connections and the location of message consumers becomes entirely opaque without requiring a name service. Moreover, new message producers and consumers can be introduced at any time. Messages are delivered efficiently to many consumers in an asynchronous way. Message Oriented Middleware (MOM) products and standard specifications like CORBA Notification Service [23] and Java Message Service (JMS) [16] support the publish/subscribe mechanism.

We use X^2TS [21] to support coupling modes at the notification service. X^2TS integrates notification and transaction services, leveraging multicast enabled MOM for scalable and reliable event dissemination. In addition, it provides the possibility to store messages in a database for event logging.

In our work, we introduced *concept-based addressing*. As its name suggests, subscriptions are made based on the concepts defined in the underlying ontology. Using concept-based addressing consumers subscribe to a concept of interest. This approach is based on the subject-based addressing principles where the subject namespace is hierarchically organized. Here concepts are mapped to the subject namespace, where the concept name is part of the subject, such as PlaceBid. It is also possible that attribute values of a concept constitute part of the namespace in order to allow a more specific subscription, for instance PlaceBid.<ParticipantId>.<ItemId>. In particular, mandatory attributes of a concept are candidates to form part of the namespace. This way, the destination of notifications is determined by self-contained information. Namespace organization is maintained in the repository.

Because concepts are represented using a common ontology, consumers do not need to take care of proprietary representations. Moreover, if concepts are extended nothing must be changed at the consumer side.

4.2 Event Sources and Adapters

An API to describe event context and to signal semantic events is provided. Based on this API event adapters convert source events into concepts based on the domain-specific vocabulary adding proper context information to support its correct interpretation. Examples of event adapters are:

- *XML adapters* that translate XML-based messages into semantic events.
- *Database adapters* that are used to signal database operations outside the database. With this purpose, trigger mechanisms can be used to detect the event and then stored procedures are used to generate the corresponding semantic event.
- *Interceptors* that are used to intercept a service request (before or after execution). Once a request is intercepted a corresponding event is signaled.

– *Alarm adapters* that play the role of a mediator to scheduled time-related events (using the alarm service) and when proper generates time-related events based on the ontology as a result.
– *E-mail adapters* that intercept e-mails according to specified e-mail properties, like sender address, subject, etc. Once caught, the necessary information is extracted from an e-mail and converted into a semantic event.

4.3 Alarm Service

This service is also considered as a source of temporal events (absolute, relative and periodic). These events require a clock scheduler in order to signal scheduled temporal events. For instance, an absolute temporal event indicating BeginOfAuction, can be defined as ”February 19, 2001 at 9:00AM GMT-5” which means that at the specified time it must be signaled. This service can also be useful for the infrastructure itself, e.g. in a distributed environment where a heartbeat mechanism is needed to produce (and consume) heartbeats within a determined periodicity. It is important to notice that in this kind of scenario, clocks used by these services must be synchronized with the timestamp service explained below.

4.4 Time (Timestamp) Service

Timestamps allow events to be ordered. This order takes a fundamental role in event consumption (e.g. chronicle) and when using time-related event operators. Depending on the given system environment, e.g. distributed or centralized, different timestamp models can be used. For example, for centralized environments a simple timestamp mechanism may be sufficient since only one clock is used to timestamp events. For distributed closed networks the 2g-precedence model may fit. For open distributed environments the accuracy interval model [20] can be used. The implementation of a timestamp service should explicitly declare all assumptions made. For this reason, a timestamp ontology is required to describe semantic assumptions about the timestamp mechanism like observation (detection time, occurrence time, reception time), clock source (local, remote, local synchronized), clock granularity, etc. The timestamp ontology is important to determine the compatibility of timestamp mechanisms.

4.5 Filter Service

Filters select notifications by discarding events based on predicates defined on event attributes. For instance, events coming from a particular source, or events which contain prices over US$ 100 are not of interest so they can be filtered and do not reach the consumer. Because events and notifications are represented using concepts from the common ontology, event filters can be specified at a domain-specific level, and are independent from source-specific representations.

4.6 Complex Event Detection

This service must be configured to recognize complex situations based on primitive and composite events. Situations of interest are described using event operators. There are several aspects that are involved when recognizing complex event situations such as those summarized in [1, 28]. In particular, the inherent characteristics imposed by

large-scale distributed environments have to be considered: the partial order of events, transmission delays, and possible failures at the sources or in the communication channel. The basic infrastructure of the complex event detector service is based on the idea of components and containers. Components are the event operators (also named compositors here) that are plugged into the container. The container itself is the complex event detector kernel which controls the event detection process. This approach avoids hard-wired event operators within the complex event detector, plugging-in only the set of operators related with the specified rules, and it provides a flexible framework to define other event operators. Compositors subscribe to primitive events or other compositors according to the complex event definition. Event instances are propagated to subscribers using the notification service described in Section 4.1. Compositors can also be configured in order to apply a consumption policy of event instances (consumption mode). This depends on the implementation of basic methods that allow to order events based on the ontology-based timestamp mechanism.

4.7 Condition and Action Service

The condition service provides an interface to register the condition (predicate) of a particular rule that must be evaluated once the corresponding event is signaled. This service plays the role of a factory, instantiating a condition object, setting its pertinent properties (predicate to evaluate, drivers to allow predicate evaluation, etc.), and subscribing it to its corresponding triggering event taking into account the specified coupling mode between event and condition. Notice that different conflict resolution policies can be applied here but concurrent rule execution is assumed in order to improve scalability and performance.

Likewise, the action service provides an interface to register the action of a rule, instantiating an action object, setting its properties to receive the corresponding notification. Depending on the rule definition, condition and action objects are associated with objects that provide the means to: access databases, invoke methods on distributed objects or wrapped legacy applications, invoke transaction control operations, access messaging services (e-mail, queues, SMS, fax), etc.

4.8 Repository and Ontology Service

We use an ontology server to store and manage the common vocabulary used in our framework. This vocabulary provides the extensible domain- and infrastructure-specific description basis to which all other services refer. The ontology server provides a common access point to the vocabulary, and provides concept definitions and textual, human-readable descriptions of available concepts for interactive exploration by developers and end users. In our repository are stored: rule definitions, their deployment and configuration, event constellations, configuration of adapters, service configurations, and subject namespace organization. This repository is used for configuration and maintenance purposes.

5 Conclusions and Future Work

We presented a service-based architecture to support active functionality which is based on a common vocabulary (ontology), services, and event notifications. Ontologies are used as a common interpretation basis to enable semantically correct interpretation of

events and notifications in open heterogeneous environments. Our ontology-based infrastructure applies homogeneously the ontology approach not only to integrate events from different sources but also to support the integration among elementary services. This allows filters and operators on events to be defined at a higher-level without taking care of source-specific representation peculiarities. ECA-rule processing in our architecture is decomposed into elementary services. These services provide a very simple and generic interface, where parameters of methods are represented using the common ontology. Therefore, the flow of work through services can be easily configured – omission or inclusion of services like condition evaluation, event filtering or complex event detection is made easy. Notifications are used to carry events from their source to interested consumers (services), i.e. notifications are the means for services to interact. For this purpose, a notification service, based on a publish/subscribe mechanism using concept-based addressing, is used. In addition, this service supports event storage and different coupling modes. Because of this conceptual foundation, our architecture promotes flexibility, extensibility and integration for large-scale Internet-based applications.

We use the Java language to specify ontology concepts and their relationships, thus avoiding any impedance mismatch between programming language and ontology specification language, and allowing the shipping of ontology concepts between different platforms without further transformations. In addition to data portability, Java supports code portability which is an important issue here since rule enforcement may be necessary to run at different tiers and at different run-time configurations. For instance, to control business rules at the client, at the middle tier or at the server. Ontology support is completely implemented and many of the ontology concepts are already defined. Our implementation also includes a running notification service that supports transactions and coupling modes, and some event adapters like XML adapter, alarm, and application adapter.

Current research involves issues related with complex event detection in open distributed environments. In particular three issues are being investigated. First, consumption modes which should take into account partial order of events and how to cope with uncertainty of event order. Second, we are looking for a minimalistic set of (low-level) event operators, and based on them define domain-specific powerful (high-level) event operators. Finally, the extension of the timestamp ontology is required in order to correctly interpret and compare timestamps coming from distributed sources. Therefore, different time synchronization dimensions and event observation mechanisms must be studied and properly organized and represented. Another aspect to be studied, is the validation of different service configurations to avoid unimplementable or inconsistent services.

References

1. (ACT-NET) K. Dittrich, S. Gatziu, A. Geppert. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. In Proc. of RIDS, LNCS 985, Sept 1995.
2. J. Bacon, K. Moody, J. Bates. Active Systems. Technical Report. Computer Laboratory, University of Cambridge, Dec 1998.
3. C. Bornhövd, A. Buchmann. A Prototype for Metadata-Based Integration of Internet Sources. In Proc. Intl Conf. on Advanced Information Systems Engineering, 1999.

4. C. Bornhövd. Semantic Metadata for the Integration of Heterogeneous Internet Data (in German). Ph.D. Thesis, Department of Computer Science, Darmstadt University of Technology, Germany, 2000.
5. A. Buchmann. Architecture of Active Database Systems. In Norman Paton (editor), Active Rules in Database Systems, Springer, 1999.
6. A. Carzaniga. Architectures for an Event Notification Service Scalable to Wide-area Networks, Ph.D. Thesis, Politecnico di Milano, Italy, 1998.
7. C. Collet, G. Vargas-Solar, H. Grazziotin-Ribeiro. Towards a Semantic Event Service for distributed Active Database Applications. DEXA'98, LNCS 1460, September 1998.
8. G. Cugola, E. Di Nitto, A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In Proc. Intl. Conf. on Software Engineering (ICSE), 1998.
9. U. Dayal, et. al The HiPAC Project: Combining Active Databases and Timing Constraints. SIGMOD Record 17(1), 1988.
10. H. Fritschi, S. Gatziu, K. Dittrich. FRAMBOISE - an Approach to construct Active Database Mechanisms. Tech. Rep. IFI-97-04, Inst. für Informatik, Univ. of Zurich. 1997.
11. S. Gatziu, A. Koschel, G. von Bültzingsloewen, H. Fritschi. Unbundling active functionality. SIGMOD Record, 27(1), 1998.
12. A. Geppert, K. Dittrich. Bundling: A new Construction Paradigm for Persistent Systems. Networking and Information Systems Journal, 1(1), June 1998.
13. A. Geppert, D. Tombros. Event-based Distributed Workflow Execution with EVE. In Proc. of Middleware '98, Sept 1998.
14. R. Gruber, B. Krishnamurthy, E. Panago. The Architecture of the READY Event Notification Service. In Proc. Workshop on Distributed Computing Systems Middleware, 1999.
15. A. Hinze, D. Faensen. A Unified Model of Internet Scale Alerting Services. In Proc. Intl. Computer Science Conference (ICSC), LNCS 1794, 1999.
16. JavaSoft, Java Message Service (JMS) spec. 1.0.1, Oct., 1998.
17. H. Kopetz. Sparse Time versus Dense Time in Distributed Real-Time Systems. In Proc. Intl. Conf. on Distributed Computing Systems (ICDCS), 1992.
18. A. Koschel, S. Gatziu, G. von Bültzingsloewen, H. Fritschi. Unbundling active functionality. In A. Dogac, et. al (editors), Current Trends in Data Management Technology, IDEA Group Publishing, 1999.
19. A. Koschel, P. Lockemann. Distributed Events in Active Database Systems - Letting the Genie out of the Bottle. Journal of Data and Knowledge Engineering, Vol. 25, 1998.
20. C. Liebig, M. Cilia, A. Buchmann. Event Composition in Time-dependent Distributed Systems. In Proc. Intl Conf. on Cooperative Information Systems (CoopIS), 1999.
21. C. Liebig, M. Malva, A. Buchmann. Integrating Notifications and Transactions: Concepts and X2TS Prototype. In Intl Workshop on Engineering Distributed Objects (EDO), 2000.
22. C. Ma, J. Bacon. COBEA: A CORBA-Based Event Architecture. In Proc. of the USENIX Conf. on Object-Oriented Technologies and Systems, June 1998.
23. Object Management Group (OMG), Notification Service Specification. Technical Report telecom/98-06-15, May, 1998.
24. S. Schwiderski. Monitoring the Behavior of Distributed Systems, Ph.D. Thesis, Selwyn College, Computer Lab, University of Cambridge, June 1996.
25. P. Verissimo. Real-Time Communication. In Sape Mullender (Editor), Distributed Systems, Addison-Wesley, 1993.
26. J. Widom, S. Ceri (editors), Active Database Systems: Triggers and Rules for Advanced Database Processing. Morgan Kaufmann, San Francisco, CA, 1996.
27. S. Yang, S. Chakravarthy. Formal Semantics of Composite Events for Distributed Environments. In Proc. Intl. Conf. on Data Engineering (ICDE), 1999.
28. D. Zimmer, R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In Proc. Intl. Conf. on Data Engineering (ICDE), 1999.