Extending WS-Notification with an Expressive Event Notification Broker

Sven De Labey and Eric Steegmans
K.U.Leuven, Dept. of Computer Science
200A Celestijnenlaan, B-3000 Leuven, Belgium
{svendl,eric}@cs.kuleuven.be

Abstract

Web service applications integrate explicit client-service interactions with notification-style programming. Such event-driven interactions are often called implicit invocations because the notification message is not in itself a business interaction. Rather, it is used to implicitly trigger such interactions as a means for reacting to events. Web Services specifications such as WS-Notification have been developed in order to standardize the way notifications are sent, but they only support the most basic form of event notification.

In this paper, we discuss the architecture and the implementation of an expressive notification broker based on extension mechanisms presented by the WS-Notification specification. This broker improves the standard WSN broker in three ways. First, it supports event correlation and hence allows for composite event detection. Second, it integrates events and reactions into Event-Condition-Action (ECA) rules that can be registered directly at the broker. Third, it introduces event-driven lifecycle management for ECA rules, thus allowing to dynamically adapt these ECA rules to changing situations.

1. Introduction

Most API's, libraries and tools for implementing Web Service applications, as well as numerous specifications aiming at standardizing web service interactions, primarily focus on *explicit* web service invocations. An interaction is called "explicit" when a client service *directly* invokes an operation published by the interface specification of the target web service. An example of an explicit service interaction is the invocation of the `print` operation, exported by the `PrinterService` port type of some WSDL description. Most research about Web Services engineering has been focusing on this explicit message paradigm, and even business process modeling, typically situated at a somewhat higher level, primarily focuses on explicit interactions between web services.

The complement of explicit invocation is *implicit invocation* and it recently gained a lot of popularity in the context of Event Driven SOA. This second interaction strategy is concerned with *reactions* rather than explicit *actions*. A Web Service that relies on implicit invocation registers its interest in certain *event types* (or groups of related events, categorized into *event channels*) at one or more *event producers*. These producers will in turn notify all interested Web Services about the occurrence of an event. This communication paradigm is termed *implicit* because the notification producer only signals the event occurrence to registered consumers. The decision about what *action* to take, is left entirely to each consumer separately. Most notification messages are followed by explicit service invocations representing the reactions of the consumers being notified. Rerouting print jobs on being notified that the printer service is stuck in a paper jam, is an example of an explicit interaction (job rerouting) following an event notification (a printer problem). So implicit invocation largely depends on explicit service invocation, but the reverse is not true.

To accomodate this need for event-driven Web Services architectures, new specifications have been developed, grouped under the WS-Notification (WSN) umbrella specification. This specification comprises (1) WS-BaseNotification [22], (2) WS-BrokeredNotification [23], and (3) WS-Topics [24, 34]. The most obvious need for specifications imposing standardized interaction patterns for implicit service interactions can be found in applications using the *Web Services Resource Framework* (WSRF) [11]. This framework was built as a refactoring of the Open Grid Services Infrastructure (OGSI) specification [30] for grid services [18], and accomodates the need for building Web Services that wrap or describe *stateful resources* using the WS-Resource specification. State changes to such resources are typically signalled using the WS-Notification standards. No surprise, then, that WSN is in itself an important and often-used part of the WSRF framework [11].

In this paper, we focus on the architectural design and implementation of an expressive notification broker for WSN-compliant web services. This broker conforms to the

¹Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders.

WSN specifications, but it augments their applicability by using various extension points that were provisioned by the specification itself. Our broker extends the atomic event notification system of WS-Notification with *complex event correlation* based on a plugin infrastructure for installing customized query languages, possibly with domain-specific composition operators. Our broker also supports the definition of *Event-Condition-Action* (ECA) rules and allows system events to influence the lifecycle of these ECA rules.

The remainder of this text is structured as follows. Section 2 gives a quick overview of the WSN standards and points out the weaknesses on which we want to focus. Section 3 shows the architecture of our WSN-based broker supporting complex event detection. Section 4 explains how this broker is extended with support for Event-Condition-Action rules. Section 5 adds lifecycle support to these ECA rules and allows lifecycle changes to be triggered by complex events. Finally, Section 6 presents related work, and Section 7 concludes.

2. Web Service Notification

This section provides a brief overview of the WS-Notification standards (Section 2.1) and shows their support for *event filtering*, an important technique to downsize unsolicited notifications (Section 2.2). This analysis allows us to point out some key weaknesses regarding the expressiveness of event filtering (Section 2.3).

2.1 Notification Standards

WS-Notification is the name for a group containing three standards, each of which is geared towards standardizing message exchanges for event notification in Web Services architectures:

- **WS-BaseNotification** [22]. This specification standardizes the way in which notification consumers register their interest for certain events at the notification producer. There is no intermediary between both parties, so the producer is responsible for (1) managing registrations and (2) sending out notifications to all interested parties. This strategy puts high pressure on the producer and is therefore only applicable in small-scale web service architectures.
- **WS-BrokeredNotification** [23]. This second specification augments WS-BaseNotification with an intermediary called the *Notification Broker*. The broker is a central service that manages consumer registrations on behalf of the producers. It also forwards notifications to interested consumers, as such further alleviating the task of the notification producer. This specification is most relevant for the remainder of this text.

- **WS-Topics** [24]. This specification standardizes the way in which notification topics are constructed. A topic compares to a CORBA *event channel* on which related events are published [7]. The specification allows topics to be structured hierarchically into *topic trees* that can be grown dynamically, as such supporting the runtime extension of *notification channels*.

The major strength of the WS-BrokeredNotification specification is that consumers and producers following WS-BaseNotification need not be adapted before they can interact with the broker. This is because the WSN Broker behaves as a notification *producer* to notification consumers, accepting consumer registrations and disseminating notifications, as shown in Figure 1. Also, this same WSN Broker acts as a consumer to event producers, registering itself with all notification producers and implementing the *notify* method from the *NotificationConsumer* interface so as to be able to receive notifications.

Figure 1. WS-BrokeredNotification example

Example interactions. Figure 1 shows a simple example of the interactions in a brokered event architecture. Two event consumers, S_1 and S_2 each register their interest in certain events e_x and e_y , respectively. This is done using the *register* method which is provided by the WSN Broker (acting as a *producer*). Some time later, a notification producer, S_3 , notifies the broker about the occurrence of event e_x . This is done using the *notify* method of the broker (now acting as an event consumer). The broker forwards this notification by invoking the *notify* method of consumer S_1 , which explicitly claimed interest in this event upon registration. S_2 is not notified because that consumer is only interested in receiving notifications of e_y .

2.2 Event Filtering

Consumers are typically interested in a subset of all notifications that the services architecture generates. This calls for a *filtering mechanism* to be integrated with the consumer subscription message. The broker, then, must only send the notification to the consumer when the associated filter expression holds. WS-Notification provides three different ways for constraining notification streams:

- **Topic expressions.** A consumer can send a *topic expression* along with its registration request to indicate what topics it is interested in. This is the preferred modus operandi when consumers are interested in *categories* of events, since topics tend to categorize notifications based on shared characteristics. An example of a hierarchical topic filter is *sports/football*.
- **Precondition.** This is a boolean expression to be evaluated against the contents of the notification message. A popular query language for expressing this type of constraints is XPath [5], though other languages can be used as well, as long as the broker knows how to process them. Any message that fails to satisfy a precondition is not forwarded to the consumer that registered that precondition [34].
- **Producer Properties.** While the two previous filters concentrate on applying constraints to the *contents* of the notification message, this third filter refers to properties about the *producer* itself. Such producer properties often refer to static properties about the service. A printer service, for instance, may expose properties such as its DPI and the number of pages it can print per minute. WSRF also adds support for modeling dynamically changing properties, for instance, the length of the job queue of a printer [11].

Figure 2. WSN Event Consumers are responsible for complex event detection and target service invocation

2.3 Evaluation

The WS-Notification specifications can be compared to publish/subscribe mechanisms such as the CORBA Event Service [26] and the CORBA Notification Service [7, 15]. But the WSN specification is arguably more expressive when it comes to event filtering, since both the notification message and the producer can be the subject of a filter expression, whereas CORBA derivatives can only constrain the message itself, based on *filterable body elements* [7, 15]. Nonetheless, WSN still lacks support for advanced features concerning implicit service interactions [35, 34] that are currently available in CORBA-based systems such as the READY Notification Service [14, 13].

In the following sections, we focus on increasing the expressiveness of the event notification system of WSN, based on the following shortcomings:

- **Query Expressiveness.** Eventhough WSN provides various ways for filtering notification streams (see Section 2.2), none of these advanced features supports filter expressions about the *correlation* between events. The specifications provide no means for expressing interest in *composite events* that are defined as combinations of atomic events; WSN treats each notification separately and does not look at the bigger picture.
- **Decentralized Bookkeeping.** One disadvantage of being unable to centrally correlate events, is that the event consumers themselves become responsible for correlating events so as to detect composite events. As shown in Figure 2, the WSN broker will forward notifications about E_1 , E_2 and E_3 to the consumer (steps 1–3), but it is up to the consumers to search for interesting patterns (step 4) before being able to react to that event pattern (step 5). This results in a high number of (unsolicited) notifications being sent to consumers and it severely complicates the development of clients, since they become responsible for the bookkeeping and correlation of very fine-grained notifications.
- **ECA Rule Support.** Active databases such as SAMOS [8] and SNOOP [4] offer a means for integrating events and actions into Event-Condition-Action rules. An ECA rule waits for an event (E) to be detected, then checks whether the given condition (C) holds and, if so, executes the accompanying *action* (A). Such ECA rules are not supported by current WSN brokers, though successful experiments show that they can be integrated with CORBA [19]. WSN's failure to support such ECA rules, however, requires an extra roundtrip from the broker (detecting the event) to the consumer (reacting to the event). Lack of a centralized ECA repository also reduces the visibility and the manageability of the relationship between *event occurrences* and the associated *reactions* because consumer registrations and consumer reactions are stored at physically different locations.

Increasing the Expressiveness of the Broker. Driven by the above shortcomings that limit the expressiveness of WSN, the remainder of this paper focuses on increasing the applicability of the WSN architecture. We do this by introducing three solutions that incrementally increase its expressiveness. First, we add support for composite event detection based on *event correlation* at the WSN broker (Section 3). This removes the need for correlating atomic notifications at the consumer side. Second, we extend this broker with support for ECA rules: we attach *conditional actions*

to complex events, thus allowing the broker to directly execute actions when it detects a composite event (Section 4). Third, we enrich ECA rules with lifecycle management, so as to allow the broker to activate or deactivate ECA rules based on events that occur in the services architecture, as such providing a means for decoupling consumers from the ECA rules they registered (Section 5).

WS-Notification Compliance. It is important to see that none of these extensions apply modifications to, or introduce conflicts with the existing WSN standards. All of our external interfaces are therefore WSN compatible, and the functionality we introduce uses extension hooks that were provided by the specification for that purpose, such as query *dialects* [23, 22]. This effectively removes the need for adapting existing WSN-driven web services when installing this extended broker. This strategy closely resembles the approach followed in other WSN extensions, such as IBM's GPASS broker [2], which focuses on the mediation and transformation of notification messages, rather than on augmenting event detection.

3. Composite Event Detection

Our architecture moves the responsibility of correlating atomic notifications for detecting composite events from notification consumers to the centralized broker. On registration, consumers describe the composite event in which they are interested by means of an *Event Query Language* (EQL) that is understood by the broker. As events get published by producers, this broker then starts correlating notifications, forwarding a newly created notification message each time the composite event occurs.

3.1 A Categorization of Events

Composite event definitions are created by combining atomic events using composition operators provided by a specialized Event Query Language. Events can be classified in three categories:

- **Atomic Events.** A notification for an atomic event signals one event that is directly issued by the Event Producer. There is no need for correlation; atomic events are immediately forwarded by the broker to interested consumers after evaluation of the optional notification message filter(s).
- **Structural Events.** This second category of events builds composite events out of atomic events and operators referring to the *position* of those atomic events in the event stream. Typical operators offered by most EQL are conjunction (e_1 and e_2), disjunction (e_1 or e_2), pre-occurrence (e_1 before e_2) and non-occurrence (e.g., *not* e_1 in [e_2 before e_3]).
- **Time-constrained Events.** Notifications referring to this kind of events combine instances of the events

mentioned above with *temporal constraints*, stating, for instance, that the composite event must occur within a time window of 30 seconds. Examples of query languages supporting these constructs are CEDR [1] and Esper [21].

WS-Notification supports only the first type of events, leading to the shortcomings discussed in Section 2.3. The implementation of our extended broker is geared towards integrating support for centralized composite event detection. Our current broker is equipped with a notification processor that correlates event queries that combine atomic events using *structural composition operators*. A consumer can register its interest in a composite event by sending an event definition comprising (a) XPath expressions matching one or several atomic WSN events and (b) structural composition operators that combine these XPath expressions based on their position in the event stream (e.g. pre-occurrence). Support for time-constrained events is not integrated by default, but can be achieved by plugging in a more sophisticated query engine, and by specifying a new *query dialect* in the SOAP registration message, as discussed further in this text.

Figure 3. Event correlation extension working on top of the WSN broker

3.2. Extended Broker for Composite Events

Figure 3 shows how we have extended the WSN Broker with a *Facade Service* and an *Event Detector*. Similar to the WSN Broker, the Facade Service offers basic WSN operations for subscribing consumers. The major difference is to be found in the *content* of a consumer registration message, which may now contain complex event queries, instead of a filter expression for atomic events. The EQL in which that event query is written is given a distinguished name that allows the broker to route the query definition to the

proper *event detection engine*. New query languages can be installed by deploying an additional event detector under a distinctive name, ed_{name} . Consumers registering event queries to be processed by that engine must use ed_{name} as the *query dialect*, an attribute in the SOAP message part that contains the event query definition. This dialect attribute is a typical extension point provided by the WS-Notification specification. It is also used by GPASS [2] to express mediation constraints.

Figure 3 illustrates the internal working of this event correlation engine is as follows. First, the consumer sends a registration to the facade service (step 1). This registration contains (1) a consumer reference, CR, to which the notification will be sent and (2) an event definition, E, describing the (composite) event in which the consumer is interested. This event definition is sent to the *Event Detector*, where it is split into atomic events (step 2). This is necessary because the WSN broker only understands atomic events. Then, the event detector, now acting as an event consumer, registers its interest at the WSN Broker for each atomic event that occurred in the initial event definition. From then, notifications of atomic events can start flowing from event producers (4a and 4b) via the standard WSN broker to our event detector (steps 4–5). The detector correlates these events according to the provided event definition E (step 6), and notifies the facade service when it detects a composite event (step 7). The facade then forwards the notification to the event consumer by invoking the `notify` method on the registered reference CR.

WS-Notification Compliance. One major design goal is the need for adhering to the WSN standard, as such avoiding the necessity to make changes to existing WSN-driven producers and brokers. Therefore, the only perceivable interaction that differs from the WSN specification is the subscription message, which now contains an *event definition* using an Event Query Language, rather than a simple topic expression for atomic events. All other interactions are standardized WSN messages. Thus, there is no need for adapting existing event consumers and event producers. Only consumers willing to react to complex events need to know about the extended subscription possibilities.

4. Combining Complex Events and Actions

Given this support for composite events based on event definitions, we now extend this architecture with support for attaching *actions* to these event definitions. This step improves the expressiveness of the broker by going from passive registrations triggering consumer notification to reactive subscriptions that are able to immediately act on composite event occurrences. Such reactive subscriptions are also known as *Event-Condition-Action* rules, where the *action* part is executed on *event* detection if the given *condition* holds. These rules stem from active databases such

as SAMOS [8], where they are used for reacting to specific database events such as CRUD operations [4].

Figure 4. Extending the correlation engine with support for ECA rules

In web services architectures, these rules are used to avoid the roundtrip to the event consumer. Rather than notifying the consumer and awaiting its reaction, the action to be undertaken is now stored at the broker, allowing immediate execution of event-handling code when the given event is detected. Our ECA rule structure closely follows the one proposed in [19] and [4], though this structure will be extended in the next Section. Their basic structure contains:

- **Event.** This can be either an atomic event or a composite event, as described in Section 3.
- **Condition.** We have defined a basic dialect for conditions, including four operators: (1) conjunction ($c_1 \wedge c_2$), (2) disjunction ($c_1 \vee c_2$), (3) negation ($\neg c_1$), and finally, (4) implication ($c_1 \Rightarrow c_2$), which requires c_2 to hold only if c_1 evaluates to true.
- **Action.** This part may contain two sorts of actions: (1) *implicit* invocations, which are notification messages to be sent to interested consumers and (2) *explicit* invocations, i.e. messages that trigger business methods offered by other services.

Figure 4 shows how the architecture from Figure 3 is extended with a *Service Invoker* component. The major difference is the subscription message (E, C, A) , which is structured as discussed above. The *Consumer References* (CR in Figure 3) are now integrated into the action list (A): each action is associated with a target CR. The event (E) and its associated condition (C) are sent to the event detector, which works similar to our discussion in Section 3 (steps 2–6). When an event is detected, the *Service Invoker* is triggered (step 7) to iterate through the action list (A) in order to react to the event. These actions can be either *explicit service invocations* (8a) or composite notifications,

i.e. invocations of the `notify` method of notification consumers (8b).

5. Event-Driven ECA Lifecycle Management

Section 4 presented ECA rules as reactive subscriptions that were activated automatically after being uploaded to the broker. This section finetunes the *lifecycle* of ECA rules. We refine the structure of ECA rules so as to allow both *consumers* and (composite) *events* to trigger ECA lifecycle changes. First, we introduce a lifecycle for ECA rules (Section 5.1). Section 5.2 then shows how our broker architecture supports lifecycle changes based on both implicit and explicit interactions.

Figure 5. Basic ECA lifecycle and state transitions

5.1. ECA Lifecycle

Initially, a web service creates a new ECA rule before uploading it to our broker. This rule can then be registered by interacting with the ECA Facade, which returns an endpoint reference (EPR) to an ECAManager resource according to the WS-Resource specification [11]. This registration fires a transition to the *suspended* state, as depicted in Figure 5. The EPR of the ECAManager resource can be used to invoke the `activate` method for triggering a transition from *suspended* to *activated*, as such activating the reactive subscription. The manager can also be used to *pause* the subscription by invoking `deactivate`, which triggers the reverse transition in the lifecycle diagram. Finally, the ECAManager resource can be used to *remove* the ECA rule, causing it to reside in the *uninstalled* state.

The use of an ECAManager is a way of manipulating the lifecycle through explicit client-service interactions based on the *implied resource pattern* [11]. For example, a consumer willing to temporarily deactivate its ECA rule sends the `deactivate` SOAP message to the ECAManager that manages the particular rule. The next Section shows how the same effect can be achieved by relying on composite events.

5.2. Event-Driven Lifecycle Changes

Making the consumer responsible for managing the lifecycle of its ECA rules may be cumbersome, especially when the ECA rule must continue to exist after the consumer service has left the architecture or when lifecycle transitions are event-driven, i.e. when the occurrence of certain events necessitates the activation or deactivation of some ECA rules. Such situations can be effectively supported by integrating *event-driven lifecycle management* in the structure of an ECA rules.

Figure 6. Event-Driven ECA lifecycle management based on ECA rule nesting

ECA rule definitions are extended with three *optional hooks*: activation, suspension, and removal. Each hook can be connected to another ECA rule. If the hook is empty, then the lifecycle transition can only be carried out by explicit consumer requests using the ECAManager. Figure 6 gives an example of a nested ECA configuration. The left ECA rule's *activation hook* is connected to a subsidiary ECA rule. If the event defined by EventDefinition of the subsidiary is detected, and provided its condition holds, then its action list is executed. One of these actions will be the transmission of the activation SOAP message to the left ECA rule.

6. Related Work

In [29], Quiroz and Panashar find that current WSN implementations such as Apache Pubsub and WSRF.NET do not provide adequate support for scalable subscription management and efficient notification dissemination. They propose a solution based on a distributed hashtable built on a structured overlay of peer nodes, where each node acts as a notification broker. Our correlation and reactive services all act as notification brokers, so our implementation can be integrated with their proposal.

Much of WSN's functionality is based on CORBA's publish/subscribe mechanism [25]. Early support for event notification was introduced by the CORBA Event Service

[26]. One problem with this specification is that it leads to a large number of unsolicited notifications. This is due to the lack of support for centralized filtering of events based on functional or QoS properties [31]. The problem is solved by the CORBA Notification Service [7], which introduces *structured events* that contain *filterable body elements* modelled as key-value pairs. Such filters cut down traffic, and hence increase the scalability of the notification system. A scalable implementation of CORBA-NS based on design patterns is proposed in [12]. Despite the advanced features of this notification broker, support is lacking for (1) complex event specifications, (2) ECA rules. The need for ECA rules, however, is provided by COBEA [20]. This architecture extends CORBA with a Composite Event Service that can detect composite events that were specified using a Composite Event Language. The functionality of COBEA augments CORBA the same way our broker augments the WS-Notification broker, although we take this process one step further, by providing event-driven lifecycle management and by supporting a pluggable event query language. The READY [14, 13] Notification Service is another example of how Event-Condition-Action rules can be integrated in a CORBA-based publish/subscribe architecture [15].

An important field of related work, though only remotely related to the domain of Web Services, is that of *active databases*. These are databases that react to (possibly complex) event occurrences based on ECA rules that were specified by the database administrator. Examples of database events include typical CRUD operations (create, read, update and delete). Successful attempts of DBMS implementations that support ECA rules include SNOOP [4, 3], SAMOS [8, 9] and COMPOSE [10]. An important difference with our work, however, is that a Web Services environment is more dynamic and heterogeneous: events originate from multiple producers and event notification is inherently a distributed operation.

ECA rules are also gaining popularity in the Semantic Web research community [27, 28], although they serve a different goal in that context. Rather than reacting to complex event occurrences signaled by heterogeneous event producers, ECA rules in the Semantic community are geared towards reacting to interesting XML patterns in RDF repositories.

GPASS [2] is a general publish/subscribe solution developed by IBM for WebSphere. Similar to our approach, GPASS is an extension of the WS-Notification standards, and it also refrains from modifying these WSN standards in order to be generally usable by Web Services using only basic WSN standards. The difference between GPASS and our notification broker is that GPASS focuses on the mediation and transformation of notification message formats. Our approach, on the other hand, focuses on extending the expressiveness of events and of the reactions to those events. Both

approaches complement each other, and integrating them should not cause major problems as both architectures behave as the Notification Broker role as defined in WSN. Intimate knowledge about the internal working on the brokers is thus not required.

The authors of WS-ECA [17] propose an eventing architecture for integrating Web Services and Event-Condition Action rules. This architecture resembles the ECA-part that was proposed in Section 4. Their architecture, however, is based on WS-Eventing, a standard that has recently been discarded in favour of WS-Notification due to severe limitations in its expressiveness, as argued in [16], a paper that provides an in-depth comparison of WS-Eventing, WS-Notification, and OGS-Notification. Our proposal, on the other hand, fully relies on WS-Notification, which has recently been accepted as the de-facto standard in event-driven web services programming. One advantage of WS-ECA, however, is that it provides a means for detecting conflicting ECA rules by analyzing the *postconditions* of each ECA rule [33]. This allows for both static and dynamic detection of conflicts in an eventing architecture. One drawback of this approach is that the authors rely on a proprietary, non-standardized specification language for defining postconditions. A similar approach was followed in [32], which extends ECA-rules with *policies*, but that work refers to CORBA architectures.

7. Conclusion and Future Work

Web services architectures are driven by business interactions that change the state of services running in the architecture, and that lead to the occurrence of events that are interesting to other services. WS-Notification is a set of specifications that tries to standardize the way in which notifications about such events can flow from the producer to interested consumers. But this specification only supports atomic events, although most interesting events typically arise out of combinations of atomic events.

In this paper, we have extended the basic WSN broker in three ways. First, we have extended the broker with an event detector that correlates atomic events based on event definitions that were entered by consumers. Second, we have attached conditions and actions to these definitions, as such providing support for ECA rules. Third, we have integrated lifecycle management for such ECA rules in an event-driven way. All these extensions are only visible for consumers willing to use this advanced functionality, while others can continue using our ECA facade without a need for code modification.

Future work comprises the introduction of a language-integrated EQL, similarly to our previous work [6] so as to increase the transparency of client-service interactions during the consumer registration phase.

References

- [1] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. *CoRR*, 2006. informal publication.
- [2] A. Bou-Ghannam and M. Roberts. GPASS: A generalized publish and subscribe solution using WS-Notification standards – <http://www.ibm.com/developerworks/websphere/>, 2007.
- [3] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617, 1994.
- [4] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.
- [5] J. Clark and S. DeRose. Xml path language (xpath) – <http://www.w3.org/tr/xpath>, 1999.
- [6] De Labey, S. et al. ServiceJ. A Java Extension for Programming Web Service Interactions. In *International Conference on Web Services. ICWS07*, 2007.
- [7] R. S. S. Filho, C. R. B. de Souza, and D. F. Redmiles. The design of a configurable, extensible and dynamic notification service. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8, 2003.
- [8] S. Gatzia and K. Dittrich. SAMOS: An Active Object-Oriented Database System. *IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Databases*, 15(1-4):23–26, December 1992.
- [9] S. Gatzia and K. R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In *RIDE-ADS*, pages 2–9, 1994.
- [10] N. Gehani, H. Jagadish, and O. Shmueli. Compose: A system for composite event specification and detection. In *Advanced Database Concepts and Research Issues.*, 1994.
- [11] GLOBUS Alliance. The WS-Resource Framework – <http://www.globus.org/wsrf/>, 2004.
- [12] P. Gore, R. Cytron, D. Schmidt, and C. O’Ryan. Designing and optimizing a scalable corba notification service. *SIGPLAN Not.*, 36(8):196–204, 2001.
- [13] R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the ready event notification service. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop*, 1999.
- [14] R. E. Gruber, B. Krishnamurthy, and E. Panagos. High-level constructs in the READY event notification system. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for Composing Distributed Applications*, pages 195–202, 1998.
- [15] R. E. Gruber, B. Krishnamurthy, and E. Panagos. Corba notification service: Design challenges and scalable solutions. In *Proceedings of the 17th International Conference on Data Engineering*, pages 13–20, Washington, DC, USA, 2001. IEEE Computer Society.
- [16] Y. Huang and D. Gannon. A comparative study of web services-based event notification specifications. In *ICPPW ’06: Proceedings of the 2006 International Conference Workshops on Parallel Processing*, pages 7–14, Washington, DC, USA, 2006. IEEE Computer Society.
- [17] J.-Y. Jung, J. Park, S.-K. Han, and K. Lee. An eca-based framework for decentralized coordination of ubiquitous web services. *Inf. Softw. Technol.*, 49(11-12):1141–1161, 2007.
- [18] K. Czajkowski, D. Ferguson, et al. From open grid services infrastructure to ws-resource framework: Refactoring & evolution – www.globus.org/wsrf/specs/ogsi_to_wsrf_1.0.pdf, 2004.
- [19] A. Koschel and R. Kramer. Configurable Event Triggered Services for CORBA-based systems. In *Proceedings of the Enterprise Distributed Object Computing Workshop*, pages 306–318, 3-5 Nov 1998.
- [20] C. Ma, C. Ma, and J. Bacon. COBEA: a CORBA-based event architecture. In *Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 9–9, 1998.
- [21] F. Marinescu. Esper: High volume event stream processing and correlation in java – <http://www.infoq.com/news/esper-esper-cep>, 2006.
- [22] OASIS Technical Committee. Web services base notification 1.3 – <http://docs.oasis-open.org/wsn/wsn-ws-base-notification-1.3-spec-os.pdf>, 2006.
- [23] OASIS Technical Committee. Web services brokered notification 1.3 – <http://docs.oasis-open.org/wsn/wsn-ws-brokered-notification-1.3-spec-os.pdf>, 2006.
- [24] OASIS Technical Committee. Web services topics 1.3 – <http://docs.oasis-open.org/wsn/wsn-ws-topics-1.3-spec-os.pdf>, 2006.
- [25] Object Management Group (OMG). Common object request broker architecture (corba) – <http://www.corba.org/>. 1991.
- [26] Object Management Group (OMG). Corba event service v1.2 – <http://www.omg.org/technology/documents/>. 2004.
- [27] G. Papamarkos, A. Poulouvasilis, and P. T. Wood. Event-condition-action rules on rdf metadata in p2p environments. *Comput. Netw.*, 50(10):1513–1532, 2006.
- [28] A. Poulouvasilis, G. Papamarkos, and P. Wood. Event-Condition-Action Rule Languages for the Semantic Web.
- [29] A. Quiroz and M. Parashar. Design and Implementation of a Distributed Content-based Notification Broker for WS-Notification. In *GRID 2006*, pages 207–214, 2006.
- [30] S. Tuecke, K. Czajkowski, et al. The Open Grid Services Infrastructure, Version 1.0 – http://globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf, 2003.
- [31] D. C. Schmidt and S. Vinoski. Object Interconnections: Overcoming Drawbacks in the OMG Events Service. *C++ Report*, 9(7), July-August 1997.
- [32] C. S. Shankar, A. Ranganathan, and R. Campbell. An eca-policy-based framework for managing ubiquitous computing environments. *Mobile and Ubiquitous Systems: Networking and Services*, 2005. *MobiQuitous 2005. The Second Annual International Conference on*, pages 33–42, 17-21 July 2005.
- [33] W. suk Lee, S. yun Lee, and K. chul Lee. Conflict Detection and Resolution method in WS-ECA framework. *Advanced Communication Technology, The 9th International Conference on*, 1:786–791, 12-14 Feb. 2007.
- [34] S. Vinoski. More Web Services Notifications. *IEEE Internet Computing*, 8(3), 2004.
- [35] S. Vinoski. WS-NonExistent Standards. *IEEE Internet Computing*, 8(6), 2004.