Implementation of a Complex Event Engine for the Web

José Júlio Alferes
CENTRIA - Universidade Nova de Lisboa
Quinta da Torre
2829-516 Caparica, Portugal
jja@di.fct.unl.pt

Gastón E. Tagni
CENTRIA - Universidade Nova de Lisboa
Quinta da Torre
2829-516 Caparica, Portugal
gastoneduardo.tagni@stud-inf.unibz.it

Abstract

One of the key aspects in the implementation of reactive behaviour in the web and, most importantly, in the semantic web is the development of event detection engines. An event engine detects events occurring in a system and notifies their occurrences to its clients. Although primitive events are useful for modelling a good number applications, certain other applications require the combination of primitive events in order to support reactive behaviour. This paper presents the implementation of an event detection engine that detects composite events specified by expressions of an illustrative sublanguage of the SNOOP event algebra.

1 Introduction

Rather than a Web of data sources, the current Web and the Semantic Web, call for a Web of Information Systems, where each such system, besides being capable of gathering information, can possibly update persistent data, communicate the changes, request changes of persistent data in other systems, and be able to react to requests from and changes on other systems. As a practical example, consider a set of data resources in the Web of music and books selling, publishers, artists, etc. It should be possible to query the resources e.g. about existing CD of artists, availability of stock in a particular seller. But in such an evolving Web, it should also be possible for a seller to announce special offers, and interesting buyer be able to detect such offers, or for instance be aware of new books by the favorite writer.

This vision of an evolving Web calls for reactive declarative languages capable of dealing with such a dynamic behavior. Recently some reactive languages have been proposed, that allow for updating Web data and are capable of reacting-to some forms of events, evaluate conditions, and upon that act by updating data [7, 5, 15, 8, 4] The common aspect of all of these languages is the use of Event-Condition-Action (ECA) declarative rules for specifying reactivity and evolution. Such kind of rules (also known as triggers, active rules, or reactive rules), that have been widely used in other fields (e.g. active databases [16]) have the general form:

on $event$ **if** $condition$ **do** $action$

They are intuitively easy to understand, and provide a well-understood formal semantics: when an event (atomic or composite) occurs, evaluate a condition, and if the condition (depending on the event, and possibly requiring further data) is satisfied then execute an action (or a sequence of actions, a program, a transaction, or even start a process).

However, in our opinion these languages fall short in what regards the kinds of events they deal with, and the way they (do not) deal with heterogeneity of data formats and languages. Regarding the former, except for [8, 4], the languages above only deal with atomic events, where these are either incoming messages or (implicit) changes of XML or RDF data. In this respect, there is some preliminary work on composite events in the Web [6], but that only considers composition of events of modification of XML-data in a single document. Detection of composite events in distributed environments has also been considered in [1]. In this system, the *Situation Manager* component is a run-time monitor that processes incoming events (produced by event sources), detects the situations to which applications must react and reports the detected situations to its clients (other applications).

Regarding heterogeneity, it is our stance that none of the languages above properly deals with it. In an open and highly distributed domain such as the Web, it is quite unlikely that there will be a unique language for expressing and detecting events, for querying data, and for specifying actions throughout the Web. Since the Web nodes are prospectively based on different concepts such as data models and languages, it is important that ECA-based frameworks for the (Semantic) Web are modular, and that the concepts and the actual languages for events, conditions

and actions are independent. Such a general ontology-based framework has been defined in [13], and the corresponding language markup, communication and rule execution model in [12][1]. In this framework, different languages for specifying and detecting events, for querying (static) data and for performing actions in the Web may be combined in ECA rules. Such a rule has an event part defined in an event language (properly identified in the corresponding rule part), a condition part and an action part. The engine associated with the framework relies on delegating the detection of events (resp. evaluation of conditions, which possibly involves gathering (static) data from the Web) and execution of actions) to appropriate event detection engines (resp. query engines, and action execution engines). The communication between the various rule parts is made by binding variables, and relies on a specific markup (see [2] for details).

In this paper we report on the implementation of a complex event detection engine that can be used together with the above mentioned general framework. As such, the event detection engine assumes that events are registered by a general ECA engine, and are marked up in a markup language that we present below. Moreover, all the communication between the event engine and its outside will be made by variable bindings, also marked up accordingly. The engine detects composite events by processing and combining atomic events. Atomic events are detected by events evaluators operating outside the system and then signaled to our system once they have been detected. Composite events are detected by considering the language operator's semantics, where the considered language is a sublanguage of the SNOOP event algebra. We use event trees to represent expressions and they are reused to exploit commonalities among several (sub)expressions.

The engine was fully implemented in Java, as a Web service. As such, though thought to work with the above mentioned framework, it can also be used by other services and frameworks, if the markup for the communication of results is respected.

The remainder of the paper is organized as follows. Section 2 describes the event engine, introduces the event algebra implemented by the detector and describes the markup language used to represent event expressions of the language. Section 3 presents the most important aspects of the event engine's implementation. Finally, in section 4 we present the conclusions and future research directions.

2 Overview of the system

The event engine presented in this paper detects composite events specified by event expressions of an illustrative sublanguage of the SNOOP event algebra [10]. Clients (ECA engines or other event evaluators) register event expressions with the engine when they want to be notified of the occurrence of events. Along with the expressions, clients define the variables (input and output) associated with the expressions they register. All the data necessary to detect an event is passed to the engine by using input variables (variable-value pairs). Output variables are used by the event engine in order to communicate the result of an event occurrence. The communication mechanism between the engine and its clients is implemented by means of these variable bindings. The event engine stores expressions and then, when a composite event is detected, the system notifies its clients of the occurrence of the event. The result of an event evaluation comprises the binding of variables (output variables) specified in the expression and the set of event instances that contribute to the detection of the event. For example, consider an atomic event *reservationCancelled(flightNumber,date,from,to,time)*; upon detection of a cancellation, the engine notifies the client and sends it the bindings of the variables specifying the number of the flight, date, etc, of the cancelled flight. Some of the variables may be bound from the start (input variables), e.g. to detect events of a pre-specified flight (number) in the example above.

The result of the bindings is marked up using a XML-based markup language. Alternative, the result of an event evaluation could be empty, meaning that the composite event cannot be detected either because a constituent atomic event will never occur or because the expression is syntactically incorrect.

Composite events are detected when their constituent atomic events are detected outside the system by atomic event evaluators and then signaled to the engine. Incoming atomic events are processed by the event engine and combined according to the semantics of the language's operators used to define the composite events. Atomic event evaluators are invoked by the event engine in order to evaluate required atomic events. The result of such evaluation is again a set of variable bindings and the atomic event instance that was detected.

2.1 Events

In our context, an event is a happening that occurs somewhere in the web, at some location and at some point in time. An event can be the insertion of a tuple in a database or the update of an XML or RDF repository. Also, events can be high-level application-dependent happenings such as the cancellation of a reservation on a flight from Lisbon to Chicago on September 17. Events are classified as *atomic events* or *composite events*. Atomic events represent those events that occur outside the system and are notified (in a push manner) to the system by messages. From the event

[1] For a more detailed description of the whole framework see [2].

detector's point of view, an atomic event is something that happens in the web and is received along with the event's information. An example of an atomic event is the *reservationCancelled(flightNumber,date,from,to,time)* event given before. Composite events are complex events specified by event expressions written in some event algebra and defined in terms of atomic and composite events. They are detected by the event engine. For example, the event *newCD(title,artist)OR newBook(title,author)* is a composite event.

2.2 Definition of the language SNOOP-R

The event detection engine presented in this paper implements an illustrative subset of the SNOOP event language. The sublanguage defined here is referred to as the SNOOP-R event algebra and includes the disjunctive operator *OR*, the sequence operator *SEQ* (also denoted as *';'*) and the conjunctive operator *ANY*. Although the expressiveness of this sublanguage is limited, it is expressive enough for investigating several aspects of events detection in distributed environments. In the following we briefly describe the semantics of these operators (composers). For a formal specification of the semantics, the reader is referred to [9].

Disjunctive operator OR. The disjunctive operator allows us to express a composite event that occurs every time one of its constituent events occurs. For example, suppose we want to be notified when either a book whose author's name is "Cortazar" or "Borges" is on sale. In order to express this we need to use an operator to combine these two atomic events. In our language this can be done by using the OR operator. Here, the time of occurrence of a composite event OR is the time of occurrence of the constituent event that has occurred. As an example consider the composite expression *OR(newBook(BT1,"Borges"),newBook(BT2,"Cortazar"))*.

Sequence operator SEQ. The sequence operator is used to define composite events where the order of its constituent events is relevant. Suppose E_1 and E_2 are events (atomic or composite). The composite event SEQ(E_1, E_2) occurs when E_2 occurs provided E_1 has already occurred. This means that the time of occurrence of E_1 has to be less than the time of occurrence of E_2. Note that, in contrast with the OR operator, composite events expressed with the sequence operator occur when all the constituent events occurs and the time restriction holds. In this case, the occurrence time of the composite event is the occurrence time of the operator's right-hand side event. As an example consider the composite expression *SEQ(newCD(CDT1,"Laura Pausini"),newCD(CDT2,"U2"))*.

Conjunctive operator ANY. A composite event $ANY(n, E_1, E_2, \ldots, E_m)$ occurs when n out of m events (distinct events) occur provided $n < m$. If $n >= m$, the composite event ANY occurs when all its constituent events occur. In this case we can write this as $ALL(n, E_1, E_2, \ldots, E_m)$. With the conjunction operator the order among the constituent events is not relevant, as long as all the required n events occur. The occurrence time of a composite event ANY is the occurrence time of the last constituent event that has occurred. As an example consider the composite expression *ANY(1,newCD(CDT1,"U2"),newCD(CDT2,"The Corrs"))*.

2.3 A Markup language for events

Events are specified by event expressions. In our case, we distinguish among three types of event expressions. Atomic, composite and opaque expressions. Atomic and composite expressions specify atomic and composite events respectively, as discussed above. Composite expressions use the language's operators to define complex events based on atomic events and previously defined composite events. Finally, in the context of our event engine, opaque expressions are also atomic, in the sense that they cannot be further decomposed in more basic events and that can also be combined to form composite expressions, but differ from atomic expressions in that they are written in some external language (understood by some other language engine). For example, an opaque expression may define an event using Java code, Prolog code, SQL code or any other language, provide that an engine for evaluating such language is known.

Evaluation of event expressions (and hence detection of composite events) relies on the exchange of expressions among different event evaluators and ECA engines. The event engine implemented here accepts and processes event expressions represented using an XML-based markup language based on the one proposed in [2, 3]. Below we describe how expressions are marked up and at the end of the section we show an illustrative example.

Atomic expressions. Atomic expressions are represented by specifying the construct's name (using the *operator* attribute), the variables declaration (*with* element), the construct's parameters (using *having* and *parameter* elements) and, the relationship between variables and parameters.

Opaque expressions. Opaque expressions are represented by specifying the language's URI, the opaque content to be evaluated and the variables declaration.

Composite expressions. Composite expressions of the SNOOP-R algebra are represented by specifying the operator's name, the language's URI, the arguments (sub expres-

sions) to which the operator is applied, the variables declaration (input and output variables) and, the operator's parameters if needed. An example of a composite expression built out from atomic and opaque expressions is presented below.

Variable declaration. Variables are represented by specifying their name, their value and their mode (to define input and output variables). Input variables are defined by setting the attribute mode's value to *use* and in this case, the variable's value must be specified (using the *literal* element). Output variables are defined by setting the mode to *bind*. Additionally, variables are related to the parameters of atomic events using the attribute *bindVar* in the *parameter* element.

```
<expression operator="OR" language="http://snoop-r.org">
 <argument name="left">
  <solve>
   <expression operator="newBook"
    language="http://ecommerce.com">
    <with> <Variable name="X" mode="use">
              <literal>HTML in 21 days</literal>
           </Variable>
    </with>
    <having>
        <parameter name="title" bindVar="X"/>
    </having>
   </expression>
  </solve>
 </argument>
 <argument name="right"><solve>
  <expression language="http://languages.org/prolog">
   <literal>
    ?- newCD(ListOfCDs), member(cd(myCD),ListOfCDs).
   </literal></expression></solve></argument>
</expression>
```

3 Implementation

The event engine's public interface includes an operation to evaluate event expressions, an operation to signal atomic event instances, and an operation to unregister event expressions. By invoking these operations, other evaluators and ECA engines can be notified of the occurrence of composite events expressed by expressions in the language, notify atomic events occurrences and unregister a previous registered expression. The system was fully implemented in Java as a Web Service. By doing this, the event detector is available throughout the Web, thus allowing for the detection of composite events whose constituent atomic events occur at different locations. In the implementation we used the API provided by [3]. This Java library implements an evaluation engine according to the framework introduced before and it provides the set of classes implementing expressions in our language as well as the web services functionality.

3.1 Event trees and event graph

In our case we follow the same approach used in other systems (see [10, 14, 17, 6, 11]) and represent event expressions using event trees. In the tree-based approach, every expression is represented as an event tree where the leaves denote atomic event types, internal nodes represent the language's operators and the tree's root is the outermost operator of the expression.

Since different event expressions registered with the event engine may use the same subexpressions and different clients may register the same expression, the detector uses an event graph in order to combine different event trees and support the reuse of common (sub)expressions. That is, if two event expressions A and B use the same subexpression C, the event tree representing the expression C is shared by the event trees representing the expressions A and B. This means that the event graph will store the information associated with the event tree being reused only once, as opposed to creating the same (sub)tree more than once. As a result of this combination, every leaf in the event graph may have several parents. The event graph is thus a collection of event trees. For example, consider the following two composite expressions where *newBook*, *newCd* and *newMovieDVD* represent atomic event types:

```
A = OR(newBook(BT,"Dan Brown"),newCD(CDT1,"U2"))
B = SEQ(newCD(CDT2,"The Corrs"),newMovieDVD(MT,"2006"))
```

Both expressions share the subexpression `C = newCD`, i.e. the atomic event type *newCD*. As a result of this, the event trees representing the expressions A and B will share the (sub)tree associated with C (a leaf node in this case). Figure 1 illustrates the event trees and the resulting event graph.

3.2 Building the event graph

When a client registers an event expression, the event engine tries to construct the event tree that represents the expression. The construction of event trees is an incremental, bottom-up procedure that starts at the leaf nodes representing the atomic event types defined in the expression and ends at the root node (an operator node). Once this process reaches the tree's root, the event engine adds the event tree to the event graph and the registration process ends.

Independently of its type, if an expression is syntactically incorrect, the event engine aborts the registration process and starts a cleaning process in order to delete incomplete event trees. Here, care is taken in order to avoid eliminating nodes that are being used by other (older) expressions, since, as mentioned above, nodes can be reused by different event trees.

Processing atomic expressions. When an atomic expression is processed, the detector checks (based on the operator's name) whether the leaf node representing the atomic event type described by the expression is already in the

graph. If that is the case, the event engine reuses the node, otherwise a new leaf node is created and added to the event tree being constructed. Every leaf node stores information about the events it represents. This information includes the variables to be bound as a result of the event occurrence (output variables), the rule or expression the event belongs to, the parent to be activated when the event occurs and an evaluation's ID (implemented as an URI) that uniquely identifies the evaluation of the expression. Every leaf may store events from different expressions (all of them of the same event type) as nodes can be shared. For example, consider the following expressions:

```
A = OR(newBook(BT,"Dan Brown"),newCD(CDT1,"U2"))
B = SEQ(newCD(CDT2,"The Corrs"),newMovieDVD(MT,"2006"))
```

These expressions use the same atomic event type *newCD*. When the event engine receives the expression A, it creates a leaf node to represent the event type *newCD* and stores at the node the event's specific information. When processing the expression B, the event detector reuses the leaf created before and stores information about the event *newCD(CDT2,"The Corrs")*. Figure 1 illustrates how the event expressions are represented in the system. In the figure, every ellipse and dashed line from a leaf represent a particular event definition stored at the leaf. For example, $A1$ represents the information about the event *newBook(BT,"Dan Brown")*. Note that the leaf representing the atomic event type *newCD* contains the information of two events as both expressions use the event type *newCD* ($A2$ represents the event *newCD(CDT1,"U2")* and $B1$ represents *newCD(CDT2,"The Corrs")*).

Processing composite expressions. The processing of a composite expression depends on the class of composite expression being considered. Here we distinguish two cases. In the first case, if the composite expression is part of the SNOOP-R language, the event detector decomposes it into smaller subexpressions and processes them recursively. In the other case, if the expression belongs to a different language (an external expression), the event engine treats the event denoted by the expression as if it were an atomic event and so it creates a leaf node to represent the event. From the detector's point of view, external expressions are treated as expressions denoting atomic events; i.e the event engine will issue a call to another evaluator to evaluate the expression and then process the results (the variable bindings).

Processing opaque expressions. When the event detector processes an opaque expression, it creates a new leaf node, stores information about the variables used in the expression (input and output variables) and then adds the event tree (a single node) to the event graph. Since the event detector does not know how to handle the content of an opaque expression, it is unable to compare them and hence it cannot reuse previous registered opaque expressions.

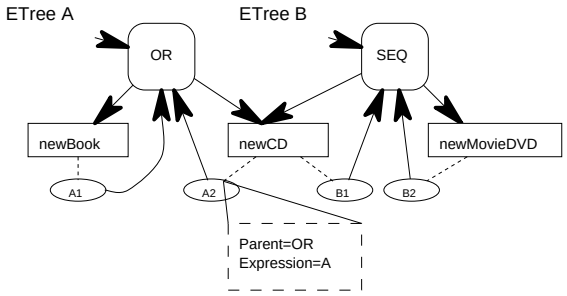

Figure 1. Representation of an event expression. Event trees and event graph

3.3 Detection of composite events

The detection of composite events follows a bottom-up process that starts when an atomic event instance is signaled to the system. Incoming atomic events are processed according to the recent context defined in [10].

When the event detector receives a message indicating that an event instance e_i of an atomic event type E_i has occurred, it computes the instance's occurrence time, stores the instance's information (variable bindings) in the leaf node associated with E_i and then activates the node. Event instances are propagated from the leaves up to the event tree's root. When an event instance reaches the root the event engine has detected a composite event.

Evaluating atomic expressions. Events denoted by atomic expressions, opaque expressions and external composite expressions are detected outside the system. The event engine evaluates these expressions by invoking another event evaluator (event provider) that is capable of detecting the events denoted by them. In order to evaluate these expressions, the detector sends to the event provider the expression to be evaluated together with the variable bindings for the input variables used in the expression. As a result of this operation, the event detector receives either an evaluation error or an evaluation ID (again an URI) that uniquely identifies the expression's evaluation. If the result of such evaluation is a not-null evaluation ID, this means that the event provider will evaluate the expression asynchronously and then communicate the results. In this case, what our event engine must do is to store this ID in order to process future results. This ID is stored at the leaf representing the event type denoted by the expression that was just evaluated. On the other hand, if the detector receives an

evaluation error, it marks the event expression as failed and starts a process to check whether the composite expression is active or not.

Activating terminal nodes. When a leaf node is activated, the event engine must validate the event instance stored at the node and then, if the validation is successful, it must propagate the instance to the appropriate parent and activate it. If an event instance is not valid, the event detector discards the instance. Here, an event instance is valid if the variable bindings associated with it are valid. In order to pass the event instance's information to the parent, the detector encapsulates the instance's information into a data structure representing the event instance. This data structure contains information about the event instance's occurrence time, the expression (or rule) it belongs to, the event type (composite, atomic or opaque) and the instance's child order to distinguish between left or right children in the sequence operator.

Validating variable bindings. Let's assume that after an event instance is notified to the system the variable bindings are available as a list in the variable `Tuple`. Let's also assume that the output variables of an atomic event expression are available as a list in the variable `FreeVars` and that the input variables associated with a composite expression are stored in the variable `Input`. Then, the variable bindings produced by an atomic event provider are valid if every variable V_i in `Tuple` is either defined in `FreeVars` or in `Input`. Additionally, every variable V_j in `FreeVars` must be defined in `Tuple`. For example, suppose that the event detector is waiting for the atomic event `newBook(Title,"Dan Brown",price)` and `Title` is an output variable in `FreeVars`. If the event instance received by the detector is `newBook("The Da Vinci Code","Dan Brown","23.90")` but no variable bindings are produced, the event engine cannot bind the variable `Title` and hence the event instance must be discarded. In the same way if the variable bindings contain something like `Tuple = [<Price,"23.90">]`, the event detector must discard the instance as the variable `Title` is not bound and it should be.

Activating operator nodes. Although the computation performed by an operator node depends on the operator's semantics, every operator must perform a series of common tasks upon activation. These tasks include storing the event instance propagated from its children, computing the variable bindings associated with the composite event and notifying composite events if needed.

Since an operator node could be reused by several event expressions, it may have several parents. Therefore, the event engine must be able to distinguish between them. I.e., how does the event engine know which parent to activate when a composite event at the node is detected? For example, consider the following event expressions:

```
A = SEQ(newMovieDVD(MvT,"Spike Lee"),E1)
B = SEQ(E2,newMusicDVD(MuT,"U2")

E1 = OR(newCD(CDT1,"U2"),newBook(BT1,"Dan Brown"))
E2 = OR(newCD(CDT2,"REM"),newBook(BT2,"Cortazar"))
```

In this case, when the atomic event *newCD("Boy","U2")* occurs, the event detector is able to detect the composite event A. Here, the detector must activate operators OR and SEQ in expressions $E1$ and A respectively. Note that the operators OR and SEQ in $E2$ and B should not be activated. In order to differentiate among parents, the event engine associates with every parent the rule (using the rule's ID) to which the parent belongs. Since every event instance contains a reference to the expression it belongs to, the event detector can easily identify the parent that must be activated. Figure 2 illustrates the approach.

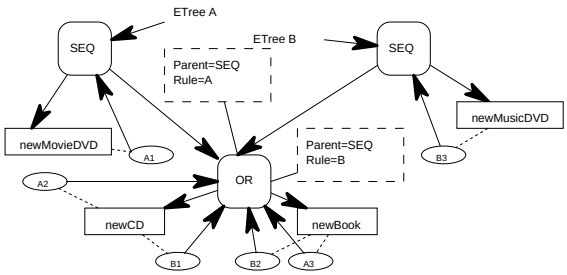

Figure 2. Schema used to identify among parents

Computing variable bindings. When an operator node is activated, the event detector must compute the variable bindings associated with the composite event being detected. That is, variable bindings from all constituents events must be joined. For example, in the expression defined below, the values of the variable `T` in both atomic events must coincide.

```
A = SEQ(newCD(T,"U2"),newMusicDVD(T,"U2"))
```

In other words, the output variable of the atomic event *newCD* is an input variable for the event *newMusicDVD*. Operators SEQ and ANY compute the variable bindings by comparing the values of the variables in the variable bindings of every constituent event (when they arrive). If any two variables whose names coincide have different values, the event detector discards the last instance received at the node. The variable bindings at the operator OR are the ones provided by the event instances that has occurred.

Notifying composite event occurrences. When a composite event is detected, the system must return the variable bindings associated with the composite event and the sequence of events that have contributed to its detection. Additionally, the event engine informs the client whether the composite event expression will produce future results or not. To notify this, the event engine returns the expression's next evaluation ID that uniquely identifies the next evaluation of the expression. In our case this value is always the expression's ID that was generated at the expression's registration time. So, if there is a future evaluation, the expression's next evaluation ID is set to the expression's ID. But, if the event cannot occur anymore (e.g. because one of its constituent events will not occur again), this value is set to `null`. For example, suppose that the composite event expression

```
A = OR(newCD(CDT,"U2"),newBook(BT,"Dan Brown"))
```

is registered in the system with ID=1 (although the ID is an URI, we use numbers for simplicity) and that at time t_1 the event *newCD("Boy","U2")* occurs with variable bindings `Tuple=[<CDT,"Boy">]` and events sequence `Literal=<expression operator="newCD"/>`. As a result of this, the event engine detects the composite event `OR` at time t_1 and returns a result where:

```
NewEvaluationID = 1
Bindings=[(CDT,"Boy")]
Sequence of events="
 <expression operator="OR"><argument>
 <expression operator="newCD"/></argument></expression>"
```

Implementing operator OR's semantics. According to the operator's semantics, a composite event OR is detected when one of its constituent events occur. When the node is activated, the event instance propagated from one of its children is stored at the node and a composite event instance is created, containing the propagated instance as its only constituent event. This composite instance is then propagated to the node's parents. Since we assume that event instances do not occur simultaneously and we are using the recent context, only one constituent event instance will be present at the node at any given time, hence the event detector keeps the instance's information in a single variable, thus reducing the memory requirements.

Implementing operator SEQ's semantics. Upon activation, the SEQ operator stores the propagated event instance. Since the operator might be shared by different expressions the event detector keeps separate storage for instances of different expressions. Moreover, since the order of the operator's constituent events matters, the event detector stores left and right children separately. When the operator node is activated, the event engine checks which child has occurred. If the right child has occurred, it checks whether the left child has occurred. If so, it compares the events' occurrence time and, if the time constraints are satisfied; it computes the variable bindings. If the left child has not occurred yet, the right child is discarded.

Implementing operator ANY's semantics. The storage requirements for this node are similar to the SEQ operator's. The only difference is that the order of the children does not matter. When the operator node is activated, the detector stores the event instance and then it checks whether the number of constituent events already detected is equals to the operator's parameter (*numberToDetect*). If so, the time restrictions are checked and, if they are satisfied, the variable bindings are computed. After this, the event detector constructs a composite event instance and activates the appropriate parent. If, on the other hand, the number of detected events is less than the number of required events, the event detector continues normally.

3.4 Deleting registered event expressions

Event expressions registered with the system can also be unregistered. An expression is deleted from the system by deleting from the event graph the event tree representing the expression. Since an event tree may reuse leaves and subtrees of another event trees, the detector must check whether a node in the event tree is being used by another tree before deleting it. In order to delete an event tree, the detector visits the tree in pre-order mode and processes every node according to its type. If the node is a leaf and is not being used by any other event tree, the event detector deletes the node from the event graph. If the leaf belongs to another event tree, it is left in the graph. In any case, the event engine deletes the expression's related information from the leaf, i.e the event expression stored at it. Now, if the node is an operator node, the event detector checks whether the node is being used by another event tree. If the node is not being used, the detector deletes the event trees associated with the node's children and then deletes the operator node from the graph. If the node is used by other trees, the event detector deletes the event trees associated with the node's children as well as any composite event instance stored at the node but, it keeps the node in the graph. For example, consider expressions A and B:

```
A = OR(newBook(BT,"Dan Brown"),newCD(CDT1,"U2"))
B = SEQ(newCD(CDT2,"The Corrs"),newMovieDVD(MT,"2006"))
```

Figure 3 illustrates how the event graph looks before and after eliminating the expression B.

4 Conclusions and future work

In this paper we have presented the implementation of an event detector engine that implements the language

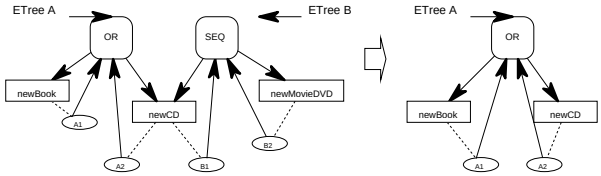

Figure 3. Event graph before and after deleting expression B

SNOOP-R, a sublanguage of the SNOOP event algebra. It detects composite events by processing and combining atomic events. Atomic events are detected by events evaluators operating outside the system and then signaled to our system once they have been detected. Composite events are detected by considering the language operator's semantics and using a bottom-up process that starts when atomic event instances are signaled to the system. We use event trees to represent expressions and they are reused to exploit commonalities among several expressions. One innovative feature of this event engine is the capability of detecting events expressed by expressions written in other languages (opaque expressions). Additionally, since atomic events are signaled to the system by using messages, they can occur anywhere in the Web. The event engine can be easily integrated in the ECA framework of [2] presented before as it implements the communication with other components by means of variable bindings.

As for future work, we would like to extend the event engine by implementing cumulative operators provided by the SNOOP language. Although the language defined here is suitable for a wide range of applications and is expressive enough to analyze different aspects of event detection, the implementation of such operators would provide a higher expressivity, extending in this way the applicability of the evaluator. Future extensions along this line could also contemplate the implementation of other parameter contexts. Another aspect that we would like to investigate is the optimization of event expressions. As pointed in [10], an event can be expressed in different ways using the language's composers. In this sense, rewriting techniques may uncover common subexpressions in the same expression or among different expressions, hence reducing the space needed in order to store them. Another aspect to consider is the variable binding mechanism. In our case, variables are bound to values without considering their types. This mechanism can be extended in order to contemplate types. Finally, we would also like to compare several techniques for event detection in the Web. The analysis of several active languages and in particular the way they employ event detectors may suggest guidelines for the implementation of more complete and powerful event engines for the Web.

References

[1] A. Adi and O. Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, 2004.

[2] J. J. Alferes and W. May. Specification of a Model, Language and Architecture for Reactivity and Evolution. Technical Report IST506779/Lisbon/I5-D4/D/PU/a1, August 2005.

[3] R. Amador. R3 Evaluation Engine Java Library. http://rewerse.net/I5/spec/r3/2005/r3.java/r3.jar.

[4] J. Bailey, F. Bry, and P.-L. Pătrânjan. Composite event queries for reactivity on the web. In *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 1082–1083, New York, NY, USA, 2005. ACM Press.

[5] J. Bailey, A. Poulovassilis, and P. T. Wood. An Event-Condition-Action Language for XML. In *Int. WWW Conference*, 2002.

[6] M. Bernauer, G. Kappel, and G. Kramler. Composite events for XML. In *13th. Int. Conference on World Wide Web (WWW 2004)*, 2004.

[7] A. Bonifati, D. Braga, A. Campi, and S. Ceri. Active XQuery. In *"Intl. Conference on Data Engineering (ICDE)"*, pages 403–418, 2002.

[8] F. Bry and P.-L. Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *20th ACM Symp. Applied Computing*. ACM, 2005.

[9] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[10] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowl. Eng.*, 14(1):1–26, 1994.

[11] M. Mansouri-Samani and M. Sloman. Gem: A Generalized Event Monitoring Language for Distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997.

[12] W. May, J. J. Alferes, and R. Amador. Active rules in the semantic web: Dealing with language heterogeneity. In *Rule Markup Languages (RuleML)*, number 3791 in LNCS. Springer, 2005.

[13] W. May, J. J. Alferes, and R. Amador. An ontology- and resources-based approach to evolution and reactivity in the semantic web. In *Ontologies, Databases and Semantics (ODBASE)*, number 3761 in LNCS. Springer, 2005.

[14] D. Moreto and M. Endler. Evaluating composite events using shared trees. In *IEE Proceedings - Software*, volume 148, pages 1 – 10, 2001.

[15] G. Papamarkos, A. Poulovassilis, and P. T. Wood. RDFTL: An Event-Condition-Action Rule Languages for RDF. In *HDMS'04*, 2004.

[16] N. W. Paton, editor. *Active Rules in Database Systems*. Monographs in Computer Science. Springer, 1999.

[17] R. J. Zhang and E. A. Unger. Event Specification and Detection. Technical Report TR CS-96-8, Department of Computing and Information Sciences, Kansas State University, June 1996.