

Filtering Features for a Composite Event Definition Language

Susan D. Urban, Ingrid Biswas, Suzanne W. Dietrich
Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-8809
susan.urban@asu.edu, ibiswas@asu.edu

Abstract

This research has enhanced a distributed, rule-based application integration environment with a composite event definition language (CEDL) and detection system. CEDL builds on existing composite event operators and selection modes, adding features to support the filtering of primitive and composite events. The filtering features includes basic parameter filtering on primitive and composite events, aggregate and quantifier filters on cumulative event parameters, and time filters for defining the lifetime of the composite event detection process. CEDL is supported by a composite event detection system that implements the filtering capabilities. This research contributes to the expression of more application-oriented events through the aggregation and correlation of distributed events.

1. Introduction

Events are increasingly being used as a communication mechanism to achieve integration in enterprise application integration and business-to-business applications, providing an asynchronous method of passing messages between modules [10]. Events can be classified as either *primitive* or *composite* events. Primitive events are simple events that mark the occurrence of some activity, such as updating data in a database, the start of a transaction, the successful or unsuccessful completion of a transaction, or a specific time of the day. Composite events are events that are created from primitive events and/or other composite events using an event algebra that specifies a logical combination of events.

Composite events are especially useful in monitoring activities in distributed applications. As an example, consider the following scenarios. A credit card company may need to keep track of the customers who don't make payments on monthly statements. The company can monitor the total amount of consecutive non-payments of a customer on continuous late payments and take appropriate action against the customers' credit rating. A banking application could restrict the number of automatic teller machine (ATM) withdrawals and total amount on the withdrawals that can be done over the

period of a working day. In an online shopping application, a company may want to be notified if the collective purchases of a customer are over a certain amount within a given period of time so that it can offer free shipping for further purchases. A common theme among all of these examples is that the generation of composite events often involves monitoring *related* events, such as late payments by the *same* customer or *aggregate* ATM withdrawals with an accumulated total from the *same* account.

The detection of composite events must therefore be combined with filtering capabilities to create a meaningful context for the use of composite events. The primary objective of this research has been to investigate the development of a composite event specification language and processing environment with filtering capabilities for the Integration Rules (IRules) [3, 13, 14] project. The IRules project is a distributed integration environment that integrates software black-box components using active rules known as *integration rules*. Events in the IRules environment provide the means for independent, distributed components to communicate with each other through the triggering of integration rules. When an event occurs, the condition of the integration rule is checked. If the condition is satisfied, the action of the rule is executed. The action can invoke services provided by distributed components or global application transactions.

The event processing capabilities of the IRules environment were originally developed in [8, 15], which included a language for the specification of primitive events, event generating capabilities for distributed components, event synchronization capabilities for synchronizing the execution of events, rules, and transactions, and an event handler for communicating the occurrence of primitive events to the integration rule processor. This research has enhanced the event processing capabilities of the IRules environment with the Composite Event Definition Language (CEDL) and corresponding composite event detection environment [1]. CEDL was developed by adopting existing event algebra operators and selection modes from past research [2, 5, 6] on composite events. The unique aspect of CEDL is the

support it provides for filtering of primitive events as well as filtering of composite events and their associated aggregate values and timelines. CEDL supports the specification of basic parameter filters on primitive events; basic parameter filters on composite events, with the ability to compare parameter values from the different events that compose a composite event; time filters that limit the lifetime of composite event detection; and indexed, aggregate and quantifier filters on cumulative composite event parameter values. The filtering capabilities are based on features initially explored in [16]. Filtering of events reduces the rule-processing load on the IRules rule manager by checking conditions on event parameters before rules are triggered. The primary advantage, however, is that filtered composite events enhance integration activities with a more meaningful approach to the expression of the types of complex, application-oriented events that are needed in the construction of distributed, event-driven applications.

Section 2 of this paper gives an overview of the primitive event language of the IRules project. Section 3 gives an overview of CEDL, while Section 4 provides an in-depth look at the filtering features of the language. Section 5 describes the implementation of the CEDL event handler. Section 6 provides a comparison of CEDL with related work. A summary of and discussion of future work is presented in Section 7.

2. Overview of IRules

As described in the introduction, the IRules project is an enterprise integration environment that integrates black-box components over a distributed network by using integration rules. The IRules environment uses Enterprise Java Beans (EJB) [4] as black-box components. Integration rules are triggered in response to events that are processed in the IRules event handler. The IRules event handler is capable of processing four different types of events: *method* events, *application transaction* events, *internal* events and *external* events. *Method* events are events associated with a call to a method on a component. *Application transaction* events are associated with the execution of global transactions. Event modifiers can be used with method and application transaction events to specify if the event is raised *before* or *after* the execution of the method or transaction. *Internal* events are events that are generated from within the black-box component. *Internal* events are therefore events that were defined in the component before the component joined the IRules environment. *External* events are events from external sources to which the IRules system can subscribe. The events in the IRules system are based on the Java Messaging Service (JMS) [7] and use the topic object as the message mediator.

The IRules environment uses the Event Definition Language (EDL) for the specification of external and

application transaction events [15]. Method and internal events are defined using the Component Definition Language (CDL) to enhance the event generating capability of the existing component [3]. Parsers have been developed for CDL and EDL to compile and store primitive event definitions into the IRules event metadata for global access by the IRules environment [8, 15].

An example of the specification of a primitive method event is shown in Figure 1. The name `afterUpdatePurchaseStatus` is the name of the method event. The event is declared to occur after the execution of the method `updatePurchaseStatus`.

```
event afterUpdatePurchaseStatus(poNo,status)
{method after updatePurchaseStatus
  (String poNo,String status);}
```

Figure 1. A Primitive Method Event Definition

3. Overview of CEDL

This section provides an overview of the CEDL operators and selection modes that have been adopted from existing research on composite events. An overview of the filtering features that have been defined as part of this research is also presented.

3.1 Event Operators

The event operators of CEDL were chosen from those of past research on composite events in active database systems and include the OR, AND, SEQ, and TIMES operators [2, 5]. The composite event `AND(A, B)` is triggered on the occurrences of event A and event B, where event A and event B represent primitive and/or composite events. For `AND(A, B)`, the order in which A and B occurs is not relevant. The event `OR(A, B)` is triggered on the occurrence of either event A or event B. Event `SEQ(A, B)` is triggered on the occurrence of event A followed by event B. The order of events is important in `SEQ(A, B)`, where the timestamp for the start of event A is older than the timestamp of event B. The event `TIMES(A, n)` is raised on n occurrences of event A, where n can be a constant integer value or '*' to represent any number of occurrences within a specified time period. `TIMES(A, n)` generates only one event but creates a collection of parameter values for each occurrence of A. This collection of event parameters can be used in the cumulative filtering capability of CEDL.

3.2 Selection Modes

Another aspect of the language design involved the adoption of event selection modes. CEDL has adopted the *recent* (*latest*) and *continuous* selection modes from [2] for use with the AND, OR, and SEQ event operators, where *recent* indicates that the most recent occurrence of an event is used in the construction of a composite event, and

continuous indicates that every occurrence of an event generates the detection process for a new composite event. The *cumulative* selection mode is automatically provided with the use of the TIMES event, where all parameter values of the same event types are formed into a collection associated with a single occurrence of the TIMES event.

Assume L: is used as a prefix on an event to specify the latest selection mode and C: to specify the continuous selection mode. Then AND(L:A, L:B) defines that both events of the AND operator have the latest selection mode. Likewise, AND(C:A, C:B) specifies that events A and B have a continuous selection mode, while AND(L:A, C:B) specifies event A with the latest selection mode and event B with continuous selection mode (with a similar meaning for AND(C:A, L:B)).

Note that every occurrence of an event initiates a composite event detector and is also possibly consumed by other composite event detectors. As an example of the semantics of AND(L:A, L:B), consider the following sequence of events: a_1, a_2, b_1, b_2, a_3 , where a_1, a_2 , and a_3 are instances of event type A and b_1 and b_2 are instances of event type B. This sequence will generate the following composite events:

- AND(a_2, b_1), since a_2 is the latest instance of A at the time b_1 occurs. The detector AND($a_1, -$) is discarded when a_2 occurs, creating a detector for AND($a_2, -$) that waits for an occurrence of B. When b_1 occurs, AND(a_2, b_1) is generated and a new detector AND($-, b_1$) is initiated.
- AND(a_3, b_2), since b_2 is the latest instance of B when a_3 occurs. The detector AND($-, b_1$) is discarded when b_2 occurs to create a detector for AND($-, b_2$), which eventually generates AND(a_3, b_2) as a composite event.

For the same event sequence, AND(C:A, C:B) will generate: AND(a_1, b_1), AND(a_2, b_1), AND(a_3, b_1), and AND(a_3, b_2). Unlike the latest mode, the continuous mode creates a new detector for each event occurrence and does not discard detectors for previous instances of the same event type. A more detailed discussion of the semantics of selection modes with all of the CEDL operators appears in [1].

3.3 Filtering Features

The third aspect of the CEDL language design involved the design of three different types of filters for primitive and composite events. The *time* filter is used to specify how long a composite event should wait for additional events after detecting the first event that it is listening for. The time filter provides a way to clear the system of composite events that may be waiting endlessly for the next event to occur.

The *parameter* filter is used to filter events based on the parameter values that are part of the event instance. This functionality is used to discard all events that do not fulfill the expected parameter values. Parameter filters

include filters on parameter values of primitive and composite events, including the comparison of parameters between the events that compose a composite event and the comparison of parameters to constant values.

The third type of filtering that is provided by CEDL is specific to the cumulative parameter values of the TIMES operator. Filtering with the TIMES operator can be done in one of three ways: the *indexing* filter, the *aggregate* filter, and the *quantification* filter. To illustrate each of the filters, assume TIMES (A (p_1, p_2, p_3), n) is defined on event A with three parameters p_1, p_2 , and p_3 . For the indexing filter, each parameter can be viewed as creating an array of values. The notation $p_1 [1]$ can be used to refer to the first occurrence of the p_1 parameter of event A (e.g. $p_1 [1] < 2$). For the *aggregate* filters, the parameter values for all of the accumulated event instances are taken into consideration. When an aggregate function like SUM is applied on parameter p_2 (e.g., $SUM(p_2) > 1000$), all of the n parameter values of parameter p_2 are added together. *Quantifier* filters also use accumulated event parameter values to perform quantification operations on the parameter values (e.g., FOR ALL p in $p_3 : p > 5$). CEDL supports both the universal and existential quantifiers.

4. CEDL Filtering Features

CEDL allows for the specification of filtered primitive and composite events.

4.1. Specification of Filtered Primitive Events

Figure 2 provides an example of a filtered primitive event. The filtered primitive event bigConsignment is based on the primitive event afterUpdatePurchaseStatus defined in Figure 1. The filter for a primitive event consists of a where clause that allows a Boolean combination of conditions on the parameters of the primitive event.

```
filter bigConsignment (status, purchaseOrderId, amount)
{afterUpdatePurchaseStatus
  (String status, String purchaseOrderId, float amount)
  where status == "complete" and amount >= 100000;}
```

Figure 2. A Filtered Primitive Event

4.2. Composite Events

Filtering features are also provided for AND, OR, SEQ, and TIMES. For simplicity, all examples assume the default use of the *latest* selection mode.

4.2.1. Filters for AND, OR, and SEQ

Figure 3 provides an example of the most basic form of the AND operator, illustrating the use of parameter and time filters. The composite keyword is used to specify the start of the composite event specification. The name freeShipping is the name of the event, while loginName is the event parameter value that will be returned with the

instance of the composite event. Composite event parameters are a projection of the event parameters of the events used to detect this event. In Figure 3, the composite event `freeShipping` is triggered to offer free shipping to customers on their next order for those who have placed two orders, indicated by a conjunction of different `completeOrder` events for the same customer within a one day period, with order amounts more than \$99 each.

```
composite freeShipping(loginName)
{completeOrder(String loginName, String orderId1,
  float amount1) AND
  completeOrder(String loginName, String orderId2,
  float amount2)
  where orderId1 != orderId2 and
    amount1 >= 99 and amount2 >= 99
  within 1 day;}
```

Figure 3. An AND Event with Parameter and Time Filters

The parameter filter on the AND event adds the criteria on the event that 1) the orders are not the same and 2) the order amount for each order is more than \$99. There is an implicit parameter filter on the `loginName` of the events, indicated by the use of the same name (i.e., `loginName`) for the first parameter of each `completeOrder` event. The implicit parameter filter implies that the orders are coming from the same customer. The time filter, `within 1 day`, adds a restriction on the amount of time the composite event will remain active in the system. The event handler will wait for 1 day after the occurrence of the first event, before it discards the event instance.

Figure 4 is an example of an OR event. In the case of an OR event, the composite event will be triggered on the occurrence of either of the two specified events. As a result, parameter filters are not supported with an OR event since the system is not aware of the event that will trigger the composite event until runtime. Time filters are also not used with OR events, since the first event that occurs triggers the OR event after satisfying the conditions in the parameter filter. In the example given in Figure 4, a composite event is defined for approving the customer order by checking the credit card information. If the event `aboveCreditLimit` or the event `cardInvalid` occurs, the event `notApproved` is thrown, which can be used to reject the customer order. The parameter `loginName` is used to identify the customer, while `errorCode` is used to specify the reason for the card rejection. In the case of an OR event, the output parameters that can be sent with the composite event are the common parameters that are defined on the events used to specify the composite event.

Figure 5 is an example of a SEQ event. This event is defined to complete an order for a customer. When the

`afterCreditCheck` event occurs for the same `loginName` and `orderId` as the `afterCheckOut` event, the composite event handler will test the parameter filter and trigger the `completeOrder` event, as long as the `afterCheckCredit` event occurs within 3 hours of the occurrence of the `afterCheckOut` event. In the case of the SEQ event, the output parameters follow the same rule as in the case of an AND event, where the parameters for both events used in the composition can be sent.

```
composite notApproved(loginName, errorCode)
{aboveCreditLimit(String loginName, String orderId,
  float amount, String errorCode) OR
  cardInvalid(String loginName, String errorCode);}
```

Figure 4. An OR Event

```
composite completeOrder(loginName, orderId, amount)
{afterCheckout(String loginName, String orderId) SEQ
  afterCheckCredit(String loginName, String orderId,
  float amount, String status)
  where amount > 0 and status == "OK"
  within 3 hours;}
```

Figure 5. A SEQ Event with Parameter and Time Filters

4.2.2. Basic Use of TIMES

Recall that the TIMES event is a cumulative event that collects event instances of the same type based on key values and the number of times that the event needs to be recorded. The composite event is fired when the event occurs the number of times specified in the event definition or when the defined time filter times out.

In the example given in Figure 6, a TIMES event `updateCustomerHistory` is defined that listens for the occurrence of `afterCancelOrder` events two times within the time period of eight weeks. This event can be used to monitor the shopping habits of customers. A '*' can be specified instead of the constant 2 to listen for an unlimited number of occurrences of a particular event in the given time span. The composite event that is listening for '*' event occurrences is triggered at the end of the time period defined in the time filter. The `loginName` in the for clause is the key for the TIMES event. The key implies that for all occurrences of the events that are being consumed, the key values are the same for all the event occurrences. In the case of a TIMES composite event, the output parameters that can be sent with the composite event are a projection of the key parameter values specified on the event.

```

composite updateCustomerHistory(loginName)
{TIMES( afterCancelOrder(String loginName, String orderId,
      float amount), 2)
  for loginName
  within 8 weeks ;}

```

Figure 6. A TIMES Event with a Time Filter

4.2.3. Parameter Filters for the TIMES Event

An example of the use of the TIMES indexed parameter filter is given in Figure 7. As seen in the example, amount(1) is an event parameter that represents the amount value in the first event instance of the composite event. Similarly, amount(2) represents the amount value in the second event instance. The example shows that the individual event parameters can be compared to other event parameters as well as to constants. In the example given in Figure 7, the parameter filter checks that orderId(1) is not equal to orderId(2) and that the value of amount(1) and amount(2) is more than \$100.

```

composite updateCustomerHistory(loginName)
{TIMES(afterCancelOrder(String loginName, String orderId,
      float amount), 2)
  for loginName
  where orderId (1) != orderId (2)
  and amount(1) > 100 and amount(2) > 100
  within 8 weeks ;}

```

Figure 7. A TIMES event with Parameter Filter

Indexed parameter filtering is not supported in the case of a TIMES event defined with a wildcard (*). In the case of a wildcard, the system is not aware of the number of instances of the event that will be captured within the given time frame.

The second type of filter provided to the TIMES event is an *aggregate* filter that can be applied on the event parameters. An example of an *aggregate* filter is given in Figure 8, where the sum function is used to determine if the sum of the cancelled orders is greater than \$3000. The various aggregate functions supported by CEDL are sum, count, min, max, and avg, where each function provides the obvious meaning. All of the aggregate functions are applied to arithmetic values, except for the count function, which can be applied to any parameter.

The third type of TIMES filter is the *quantifier* filter that is applied to the cumulative parameter values. The system provides both *universal* and *existential* quantification. An example of the universal quantifier filter is given in Figure 9. In this example the filter determines if the amount for each order is greater than \$100.

```

composite updateCustomerHistory(loginName)
{TIMES(afterCancelOrder(String loginName, String orderId,
      float amount), 2)
  for loginName
  where sum(amount) > 3000 and orderId(1) != orderId (2)
  within 8 weeks;}

```

Figure 8. A TIMES Event with Aggregate Filter

```

composite updateCustomerHistory(loginName)
{TIMES(afterCancelOrder(String loginName, String orderId,
      float amount), 2)
  for loginName
  where orderId(1) != orderId(2)
  and for all a in amount: a >= 100
  within 8 weeks ;}

```

Figure 9. Universal Quantifier Filter in a TIMES Event

4.3. Use of Nested Composite Events

Composite events can be composed in a nested fashion to create more complex composite events. The example given in Figure 10 illustrates a complex composite event that is created as a result of nesting other composite and/or primitive events. The nested composite event is monitoring possible nuisance shoppers who place orders and then either return items, register complaints on the items purchased, or cancel the order within a period of twelve weeks. The possibleNuisanceShopper composite event is defined as a SEQ event, with completeOrder followed by a disjunction of three different events (i.e., a TIMES event on returnItems, a TIMES event on registerComplaint, or a cancelOrder event). In this example, there is an implicit filter on the customer's loginName and orderId to ensure that all events are associated with the same customer and order.

Figure 11 shows an example of detecting multiple instances of a complex composite event, where nuisanceShopper is defined on multiple occurrences of the composite event possibleNuisanceShopper over a period of 24 weeks. The complex events in Figures 10 and 11 demonstrate the strength of the CEDL composite event operators together with the filtering capabilities for defining meaningful, application-oriented events.

5. Implementation of the Composite Event Detector

The IRules composite event handler is based on a binary mode of event detection similar to the Hermes [12] project. The composite event handler listens to the primitive events generated by the primitive event handler and the composite events generated by the composite

event handler as part of the composite event generation process. In [12] composite events are handled as multiple distributed objects, which are floating objects that can be used in a distributed environment where other components share the same object. Composite events are handled in a similar manner in this research, with the ability to use composite events already detected in the system. In Snoop [2], the composite event detection was handled using event graphs. Samos [5] used petri nets for the detection of composite events.

The Java Messaging Service (JMS) [7] is a messaging middleware that enables enterprise applications to asynchronously communicate with each other in a distributed network. JMS provides a point-to-point and publish-subscribe mechanism. In the point-to-point mechanism, applications send messages directly to each other. In publish-subscribe, a mediator is used to pass messages between applications, where applications interested in listening to the events register with the event mediator and applications interested in sending messages send them to the mediator. JMS uses topic objects as an event mediator. This research uses the publish-subscribe mechanism of JMS technology for event generation. The primitive event service component is based on the same technology and hence adapted into this research. A global clock was implemented in this research based on the Network Time Protocol (NTP) [19] model. Time stamping on the events is done by getting a global time from the global clock.

The IRules primitive event handler [8] sends the events generated in its system to the rule manager. The primitive event handler was modified in this research to add functionality to send the events that were detected to the IRulesPrimitiveEvent topic. The composite event handler developed in this research listens to this topic to receive all of the primitive events generated in the system. The composite event handler sends the composite events that it generates to the IRulesCompositeEvent topic and listens to the same events. This enables composite events to subscribe to other composite events. A Detect module collects event instances and combines them to create composite events. A Filter module takes these composite events and applies a filter on them as specified in CEDL. These composite events are sent to the IRulesCompositeEvent topic. The composite event handler module also generates an eventMsg object and sends it to the rule manager for rule processing.

A dedicated EventDistributor was developed as part of this research that keeps track of the event subscribers and the events that each of the subscribers is interested in. The composite event handler listens to all of the events coming from the IRulesPrimitiveEvent topic and IRulesCompositeEvent topic. The EventDistributor takes these events and forwards the event instance to only the subscribers that have subscribed for it. This mechanism

prevents the broadcasting of events on the network, thus reducing network traffic. The composite events register with the EventDistributor to listen to the events that they are interested in at system startup time and also when a new event is activated.

6. Comparison to Related Work

Of the related research examined [2, 5, 6, 11], ODE [5] and COBEA [11] are the only research on composite events that provided any significant filtering capability. In ODE, the filters are provided in the form of masks on objects, which use the object attribute values to provide the filtering. In COBEA, filters are provided by comparing the parameter values to constants, similar to the implementation in CEDL. CEDL extends that functionality by allowing comparisons between event parameters. The indexed, aggregate, and quantifier filters of CEDL are a unique feature provided on cumulative event parameters that have not been adequately addressed in past research on composite event specification languages. Distributed environments must be capable of expressing such constraints for the aggregation and correlation of events from distributed sources.

COBEA provides time filtering by providing functionality for specifying a duration object. This object has a time to start and end that tells the system when the event will be activated and when the system will stop listening for an event. ODE does not support the concept of time filters on the events. The time filter incorporated in CEDL functions in a manner similar to COBEA. The main difference in the time filter provided in COBEA is that the time period is specific. In CEDL the time filter is relative to the occurrence of the first event consumed by the system for a given composite event. The duration object in COBEA can also be applied on individual events used in the composition of the composite event. This feature has not yet been included in CEDL.

```
composite possibleNuisanceShopper(loginName)
{ completeOrder(String loginName, String orderId,
float amount) SEQ
((TIMES (returnItems(String loginName, String orderId,
String itemNo1), 2)
for loginName, orderId
within 4 weeks; OR
TIMES (registerComplaint(String loginName,
String orderId, String itemNo2), 2)
for loginName, orderId
within 4 weeks; ;) OR
cancelOrder(String loginName, String orderId) ;)
within 12 weeks;}
```

Figure 10. A Nested Composite Event

```

composite nuisanceShopper(loginName)
{TIMES (possibleNuisanceShopper (String loginName), *)
  for   loginName
  where count (loginName) > 3
  within 24 weeks ;}

```

Figure 11. Detecting Multiple Occurrences of a Complex Composite Event

7. Summary

This research has enhanced a distributed event-based integration environment with a composite event definition language and detection system. CEDL builds on existing event operators and selection modes, adding features to support the filtering of primitive and composite events. The filtering features of this research enable users to define expressive, application-oriented events that make use of aggregation and correlation of events in the event definition. The filtering features also have the potential to enhance the performance of the IRules rule processing system by only triggering events that fulfill the filtering condition. CEDL is supported by a parser that generates metadata for the description of composite events [1]. An event detection and handling module based on the work in [12] has also been implemented that composes the primitive and composite events into complex composite events [1] and implements the filtering features of CEDL.

The current research direction of the IRules project is moving towards a service-oriented architecture known as the DeltaGrid. The DeltaGrid involves the integration of Grid Services with notification capabilities. The research presented in this paper is being integrated into the DeltaGrid system to enable a composite event handling feature over Grid Services. The event handler and detection unit will need to be redesigned for greater compatibility with Grid Services technology. The event handler must also be extended to function as a distributed event handler. Performance issues related to operating within a distributed environment, such as time lag, network delays, and point of failure still need to be addressed.

8. References

- [1] I. Biswas, *A Composite Event Definition Language and Detection System for the Integration Rules Environment*, M.S. Thesis, Dept. of Computer Sci. and Eng., Arizona State Univ., 2005.
- [2] S. Chakravarthy and D. Mishra, "Snoop: An Expressive Event Specification Language for Active Databases," *Knowledge & Data Eng.*, vol. 14, no. 10, 2004, pp. 1-26.
- [3] S. W. Dietrich, S. D. Urban, A. Sundermier, Y. Na, Y. Jin, and S. Kambhampati, "A Language and Framework for

Supporting an Active Approach to Component-Based Software Integration," *Informatica*, vol. 25, no. 4, 2001, pp. 443-454.

[4] Enterprise Java Beans Specification 2.0, <http://java.sun.com/products/ejb/docs.html>

[5] S. Gatzju and K. Dittrich, "Events in an Active Object-Oriented Database System," *Proc. of the 1st Int. Workshop on Rules in Database Sys.*, Springer, 1993.

[6] N. H. Gehani, H. V. Jagadish, and O. Shmueli, "Event Specification in an Active Object-Oriented Database," *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, California, 1992, pp. 81-90.

[7] Java Message Service, Sun Microsystems. <http://java.sun.com/products/jms/>.

[8] S. Kambhampati, *An Event Service for a Rule-Based Approach to Component Integration*, M.S. Thesis, Dept. of Computer Sci. and Eng., Arizona State Univ., April 2003.

[9] J. Levine and D. Mills. "Using the Network Time Protocol to Transmit International Atomic Time (TAI), *Proc. of the Precision Time and Time Interval Applications and Planning Meeting*, Virginia, 2000.

[10] D. Linthicum, *Enterprise Application Integration*, Addison Wesley Publishing Company, 1999.

[11] C. Ma, and J. Bacon, "COBEA: A CORBA-Based Event Architecture," *Proc. of the 4th USENIX Conf. on Object-Oriented Technologies and Sys.*, New Mexico, 1998, pp. 117-131.

[12] P. Pietzuch and J. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture," *Proc of the 22nd Int. Conf. on Distributed Computing Systems*, Austria, 2002, pp. 611-618.

[13] S. D. Urban, S. W. Dietrich, Y. Na, Y. Jin, A. Sundermier and A. Saxena, "The IRules Project: Using Active Rules for the Integration of Distributed Software Components," *Proc. of the 9th IFIP 2.6 Working Conf. on Database Semantics: Semantic Issues in E-Commerce Sys.*, Hong Kong, 2001, pp. 265-286.

[14] S. D. Urban, S. W. Dietrich, A. Sundermier, Y. Jin, S. Kambhampati, and Y. Na, "Distributed Software Component Integration: A Framework for a Rule-Based Approach," *Handbook of Electronic Commerce in Business and Society*, R. Watson, P. Lowery, and J. Cherrington (eds), 2002, pp. 39T5-421.

[15] S. D. Urban, S. Kambhampati, S. W. Dietrich, Y. Jin, and A. Sundermier, "An Event Processing System for Rule-Based Component Integration," *Proc. of the Int. Conf. on Enterprise Information Sys.*, Porto, Portugal, 2004, pp. 312-319.

[16] S. D. Urban, A. Unruh, G. Martin, and M. Modine, *Expressing Composite Events in InfoSleuth*, MCC Corporation, Tech. Report #MCCINSL, 1998, pp. 131-98.