Extending a Knowledge-based Network to support Temporal Event Reasoning

John Keeney, Clay Stevens, Declan O'Sullivan
Knowledge & Data Engineering Group & FAME,
School of Computer Science & Statistics, Trinity College Dublin, Dublin, Ireland.
{John.Keeney | Declan.OSullivan}@cs.tcd.ie, cstevens@tcd.ie

***Abstract*—While the polling or request/response paradigm adopted by many network and systems management approaches form the backbone of modern monitoring and management systems, the most important and interesting events, faults, alerts and log messages arrive at the management agent in a push-based asynchronous manner. However, in the management infrastructure itself, at the point where events are initially processed and matched to subscribers, there have been few attempts to identify relationships or dependencies between events. This means that most of this burden is placed on the management application, or indeed the managers themselves. This research investigates enhancing the expressiveness of a knowledge-based networking middleware with the addition of three temporal operators to be used in subscriptions to select matching events. A prototype design is presented and a number of implementations are compared. The approach is also motivated using two scenarios for temporal correlation of warnings and faults in managed networks. The effect on the scalability of the extended knowledge-based network system is also evaluated.**

I. Introduction

As network technology rapidly develops, and enterprises and network providers place ever increasing demands on their networks, current network scales and complexity are increasing dramatically. However, traditional event and fault management approaches involve inflexible and rigid hierarchical manager/agent formations, relying on significant human intervention and analysis, which become increasingly difficult as scale and complexity grows.

Increased reliance on asynchronous event-based systems requires more real-time, complex event processing to filter and analyze event streams to provide relevant information to managers. If performed at the application level, this event processing is expensive, both in terms of the computation required and the cost of existing commercial solutions (such as SMARTS InCharge [1] and HP OpenView [2]). New solutions are required to make complex event processing affordable and easy, while maintaining the expressiveness and scalability of other distributed, event-based systems. In the following sections we present an extension to a semantically enhanced event middleware to support complex event processing and temporal reasoning. We also evaluate the scalability of our approach, and explore real-world system management deployment scenario for the resulting system.

II. Background and related work

The earliest distributed event distribution networks were usually channel-based, where messages published into a specific channel were delivered only to users who placed a subscription request for that channel, or subject- or topic-based, whereby subscriptions were made based on a well-known and structured set of subject attributes [3], as labelled in the message header information. These systems, although easily scalable and simple to use, require client applications to do most of their own event filtering.

Later systems (e.g., Gryphon [4], Siena [5]) used a content-based approach, where subscription filters could be specified over the entire contents of the event messages themselves, or type-based approach (Hermes [6]) where event messages are strongly-typed data objects. These approaches allowed more expressive filters, but also increased the complexity of message routing [3]. Recent research has focused on improving scalability in content-based publish/subscribe (CBPS) systems, improving particularly the routing and subscription matching algorithms used to filter and deliver messages [7][8][9]. Other work has improved the expressiveness and flexibility of the subscription mechanism so subscribers can have more fine-grained control when specifying the types and contents of the events they are interested in [10].

In particular, recent research projects have added support for semantic data filtering in the event matching and delivery middleware (e.g. [11][12][13][14]), where this approach is specifically termed *knowledge-based networking* [10]. This approach allows for very flexible subscriptions by additionally using rich, structured, semantic data drawn from application-specific ontologies, while at the same time maintaining efficiencies developed for content-based systems [10].

Knowledge-based networking approaches have been extensively applied to the fields of self-managing networks [13][14], autonomic communications [15], context distribution [13][14], semantic interoperability [14], and fault management [13][16]. However, with the exception of [16], this approach adopts a pure event-by-event approach, with no regard for relationships or dependencies between different events. The approach adopted in [16], which motivates parts of this work, provided some support for a strict sliding-window based correlation of events based on semantic causal relationships, however, much of the correlation was performed at the application level rather than in the middleware.

The message filtering provided by a content- or knowledge-based system could be adapted towards complex event

processing (CEP). However, CEP systems filter a continuous event stream and detect composite events, which are composed out of simple (atomic) events based on the relationships between those events, either logical or temporal. The detected events are then correlated by the system to determine causal or spatial relationships between the events and to trigger certain responses from the client system. Within CEP, specialized query languages called event processing languages are used to filter events from the event stream. These languages are generally quite capable of quickly filtering extremely large quantities of events very quickly (over 50,000 events/sec in the case of SASE [17]) using query languages over the event stream that operate in a similar manner to SQL (by means of selecting events that match certain constraints). However, the predicates available are limited to such operations as can be easily expressed in an SQL-like query language.

Composite events are compositions of simple or atomic events joined by logical or temporal operators which generally express conjunction, disjunction, sequence, or repetition. These systems can be divided between two main groups: active databases (e.g., SAMOS [18], Ode [19]) and middleware systems (e.g., PADRES [20], READY [21]). Those divisions can each be further sub-divided between systems which use a tree- or graph-based detection algorithm and those which use finite-state automata. Active database systems generally employ event-condition-action rules by which they respond to data manipulation events (such as INSERT and UPDATE queries) which meet certain conditions by executing specified actions to further manipulate the data in the database. However, this limits the active databases to responding to data manipulations rather than arbitrary events. Middleware systems get around this limitation by responding to any events that get published to them, but either sort of matching algorithm (graph- or automaton-based) requires the implementing system to continually be engaged in a form of forward matching which consumes resources by forcing the brokers to preserve the portion of each composite subscription which has been matches as state at the broker.

Our work validates a new approach to composite event detection in complex event processing: that of detecting composite events through predicate matching over historical event data. This work exploits the functionality provided by semantically-enhanced publish/subscribe systems while supporting the requirements of complex event processing. This means that we are not limited to an active database system (and thus rely on the operations allowed by the Event-Condition-Action (ECA) rules), nor do we sacrifice the computational resources necessary to save state for forward matching (as in existing middleware systems).

III. Temporal Reasoning

The majority of time-based reasoning in event processing systems to date is based on the work by James F. Allen in the early 1980s [22]. Allen introduces the concept of representing temporal intervals, as opposed to earlier work which used only point-based representations of events. Using these intervals, Allen identifies a total of thirteen unique relations which can exist between two intervals, X and Y , consisting of seven base relations and their inverses, as represented in figure 1 below.

These interval relations can also be described based on a comparison of their start and end points represented numerically. The interval relations as translated into numeric expressions, also listed in figure 1 below (X_S represents the beginning of interval X, and X_E represents its end).

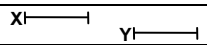
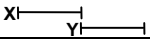
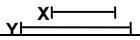
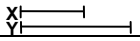
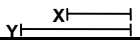
Interval Relation	Representation	Operator
X before Y		Y_S AFTER X
X equals Y		Y_S WITH X, Y_E WITH X
X meets Y		Y_S DURING X
X overlaps Y		Y_S DURING X, Y_E AFTER X
X during Y		X_S DURING Y, X_E DURING Y
X starts Y		Y_S WITH X, Y_E AFTER X
X finishes Y		X_S DURING Y, X_E WITH Y

Figure 1: Allen's interval relations as time point operations

We consider Allen's interval relations as they relate to comparing a single time stamp (either the start or end time of one interval) to another interval. Using this view, each of Allen's seven intervals can be represented using three operators representing the relationship of a point P (either a start or an end) to an interval X: AFTER, WITH, and DURING, also shown in figure 1. These three operators will be discussed in more detail in the later sections.

One of the main criticisms of such a point-based algebra is that it sometimes does not correspond exactly to real-world events, which can sometimes have uncertain beginnings and endings rather than explicitly-declared and well-defined end points. While some approaches utilize fuzzy temporal relations, Allen's original description of the relations is still very intuitive and serves as the ultimate basis for the temporal relations used in most event-based systems.

IV. Design

To support composite event detection in a knowledge-based system, we add a persistent data store component to store historical event data, and a set of temporal operators for matching subscription filters against the historical events in the data store using Allen's interval relations.

This approach differs from existing systems which use active databases instead of distributed publish/subscribe systems or else detect events using tree- or automaton-based forward matching. We chose to extend our knowledge-based publish/subscribe mechanism for three reasons: The active database systems reviewed are too restrictive on the types of events used in the ECA-rules; Using a tree-algorithm or a finite state machine for forward pattern matching requires the

system to save the state of each partial match for each composite subscription at each broker which greatly increases the complexity of the matching algorithms the broker must invoke for each publication; and finally, the semantic capabilities of knowledge-based systems (which are lacking in other approaches) are more expressive to the sorts of events which could be detected in complex event processing.

A. Data store design

In order to store historical events for future comparison using the proposed temporal operators, we add a data store component to each broker of the original knowledge-based system. This component allows each broker to store, retrieve, and analyze publications as they pass through the brokers. In particular, the data store performs three major functions: Storing (or updating) uniquely-identifiable event publications in some persistent data store; Retrieving specific event publications for inspection by the system; and Checking to see if the data store contains information about any events which match a particular semantic- or content-based subscription filter containing other arbitrary constraints (such as checking the value of certain attributes).

B. New temporal subscription operators

As shown in figure 1 above, each of Allen's temporal intervals can be represented using three operators AFTER, WITH, and DURING. The three operators all operate by comparing a reference time X_R, either the start or end of one interval X, to a target time Y_T from the second interval Y. For the AFTER operator, the reference time varies between X_S (start of X) and X_E (end of X), but the reference time is always the end of Y (Y_E). The DURING operator actually compares X_R to both the start and end of Y, giving it two target times. Finally, the WITH operator compares X_R to the corresponding field of Y, such that R=T (i.e. X_E with Y_E, or X_S with Y_S).

In order to use the three operators listed above, we first need to represent them as predicates taking a *reference attribute* (taken from the event publication – start or end time) as the reference time and a *target filter* (which selects events from the data store) which returns events corresponding to the target interval. For DURING and AFTER, we use a direct translation of the reference times into attributes, using the attributes X_S and X_E to represent the start and end time of a publication X respectively, and a filter F as the target. An operator applies if there exists a stored publication, Y, such that the relation defined by the operator in the filter holds between the reference attribute and the start and end time of the event represented by Y. For performance tuning, the AFTER operator accepts an additional time limit, L, which defines the size of the sliding window between the end time of the compared event and the reference time.

The WITH operator checks if the start or end time attribute in one publication is *equal* to the corresponding attribute in another. The WITH operator, however, can be made more general when applied to content-based publication (event) attributes. It can compare the value of an arbitrary attribute (not just start or end time) of a publication X to the value of the same attribute in any matching stored publication, Y. By generalizing the equivalence test from WITH into an arbitrary operator, we represent the WITH operator described above with a general FILTER operator that can compare any arbitrary attributes using an arbitrary operator. The FILTER operator also accepts a configurable limit number of historical events to check when looking for a match.

The three operators added are summarized in Table 1 below (where F(Y) means that the publication Y matches the filter F, and X_R represents the reference attribute).

Operator	Condition
X_R **DURING** F	$\exists Y : F(Y) \wedge (Y_S \leq X_R) \wedge (Y_E \geq X_R)$
X_R **AFTER(L)** F	$\exists Y : F(Y) \wedge (Y_E < X_R) \wedge (Y_E \geq (X_R - L))$
X_R **FILTER(OP, L)** F	$\exists Y : F(Y) \wedge (X_R \text{ OP } Y_R)$

Table 1: Temporal Operator Summary

Together the AFTER, DURING and FILTER operators not only represent all seven of Allen's interval relations, but by virtue of the generality of the FILTER operator (as a generalization of the WITH operator), allows for even more applications of the operator than the temporal operators considered here.

Performance wise, the data store must iterate over all of the events which initially match the time constraints provided by the temporal operators. This means that the time taken to process each operator should grow as more event publications are retrieved from the data store to check stored historical events against the currently processed event (up to the limit L configured in the operators themselves). It is feasible to implement this design in such a way that the processing time grows no worse than linearly with the number of events returned from the data store.

As a further benefit, this design could also be employed on any publish/subscribe system, not just a knowledge-based networking system, as long as the system can pass a filter as the target of the new operators. The actual time values used by the system do not matter (as long as they are fully-ordered and defined for each publication), while the design of the data store component is simple enough to be implemented any number of ways, either with a relational database system or even using flat log file storage.

V. Implementation

Our implementation is based on a Java-language implementation of a knowledge-based networking system ("**KBN**") described in [10] which itself was implemented on top of the hierarchical Java version of Siena [5]. All extensions previously added (up to the KBN version in [10], and this extension) were designed to appear very similar to and backwards compatible with the Siena Java API [5].

The implementation relies on a number of assumptions about the use of the system, specifically relating to the timestamps: Event start and end timestamps are totally-ordered and defined by some external source; Event timestamps are to be delivered as content-based publication

attributes; The end time of a particular event will always be greater than or equal to the start time of that event; and Any publication marked to represent the end of an event will be preceded by a publication marked to represent the start of that same event (excluding instantaneous events).

Furthermore, as a general maxim in content-based systems is to minimize the amount of meta-data attached to each publication, and instead carry such data in the open payload, we have implemented our design using a number of conventions which allow it to utilize the existing Siena framework in a consistent manner. In particular, our implementation uses four specific message attributes, as shown in table 2 below, to represent the start and end times of our events, as well as the unique identifiers for the events, and some additional flags to denote whether a publication represents the start or end of an event (or an instantaneous event, in which the start and end time are equal).

Attribute Name	Attribute Type	Description
PUB_ID	String	Event UUID
KBN_PUB_TYPE	Bag	Publication Type
KBN_START_TIME	Long	Event start timestamp
KBN_END_TIME	Long	Event end timestamp

Table 2: Special temporal attributes

PUB_ID is a string representation of a universal unique identifier used to tie start and end publications together. KBN_PUB_TYPE denotes which time marker the event represents, the start or end time, or if the event is instantaneous, using one of three values: START, END, or INSTANT. KBN_START_TIME and KBN_END_TIME store the timestamps for the start and end of an event, respectively. These two attributes are used as the reference attributes for the AFTER and DURING operators.

A. Data store implementation

For our implementation, we decided to implement the data store component in three different ways using two commonly-used relational database systems, Oracle and MySQL. These were chosen because of their widespread use, their efficiency in storing linked data (such as attributes linked to a publication), and to explore some of the more advanced features (including some semantic functions) in Oracle 11g.

As described, the data store is used to store historical event publications and used by the event broker to compare newly received events to stored events, subject to both a content-/semantic-based subscription filter (applied to each candidate publication) and a temporal operator to compare matching publications.

For each publication the database driven data store stores the PUB_ID, KBN_START_TIME, KBN_END_TIME in one table, with each additional named and typed data attribute contained in the publication stored in another related table.

In addition to methods to store and retrieve individual publications, each data store implementation has a hasMatchingPublication() method. In its simplest form this method takes a (non-temporal content-/semantic-based) subscription Filter as an argument and requires the data store implementation to check the passed Filter against the stored publications in order to determine whether the Filter matches any stored historical publications. This operation can be implemented in a number of ways depending on the capabilities of the underlying data store, and so could potentially benefit from optimizations in the implementation.

1) MySQL data store

This implementation takes a very naive approach to finding matching events from the database, using simple SQL queries to find any/all events which match the temporal constraints imposed by the operators, returning them into the KBN broker's process, and looping through them in code to test against the passed-in filter using the standard publication matching functionality available with the KBN codebase. All connections to the database were implemented using pooled localhost JDBC connections. As such, the MySQL implementation is expected to scale linearly with the size of the result set. In addition to the content-based publication matching performed by the KBN codebase, semantic-based publication matching is in the KBN handled using the Jena [23] framework.

2) Simple Oracle data store

The simple Oracle data store implementation was implemented exactly as per the MySQL data store. It uses just the standard RDBMS features of Oracle 11g. It was intended as a comparison to the following Advanced Oracle data store.

3) Advanced Oracle data store

The Advanced Oracle data store was implemented to test some of the advanced features of the Oracle 11g database system. These features include ontology-extended relational queries introduced as part of Oracle's Semantic Technologies [24] and the ability to invoke static Java methods as stored procedures from the Oracle 11g instance [25].

In order to perform the matching of stored events using the hasMatchingPublication() method the advanced Oracle data store invokes an internal stored procedure MATCH(). This stored procedure is actually a static Java function (based on code in the KBN codebase) which parses the filter passed into hasMatchingPublication() to perform the (non-temporal) filter matching *inside* the database. Any publication which matches the filter is then included in the result set and returned from the database to be compared using the temporal operators. As this greatly reduces the size of the result set returned from the database and there is no external looping required to analyze these results, we expect that this implementation would perform rather better than the simpler MySQL and Oracle implementations, perhaps scaling against the number of publications at a better than linear rate.

Performing the filter operations passed into the hasMatchingPublication() method of the data store implementation *within* the database makes the logic of the data store much simpler, as the query only returns rows which are already checked against the filter, but it adds some other technical complications. The Oracle Java VM is run in a different process than the rest of the KBN broker, so none of the objects or configuration of the KBN broker process can be

accessed directly by the matching class. In order to perform the content-/semantic-based matching internally a large proportion of the KBN codebase must be loaded by the database when the MATCH function is invoked.

As mentioned Oracle 11g also includes support for semantically-enhanced database queries (as an extension to standard relational DB queries) by comparing against a loaded and reasoned ontology stored in the database. This means that the ontological operations required to support semantic-based subscription filters in the KBN implementation can be handled by the database rather than using the Jena framework as before. This means that the incorporation of an Oracle-based data store actually simplifies the implementation of the non-temporal semantic operations of the KBN as all semantic/ontological queries can be handled by the data store.

In order for the stored MATCH procedure invoked in the database to match nested temporal or filter operators (or ontological operators), the matching procedure must itself sometimes invoke the data store to find matching publications or perform other queries. Restrictions placed on stored procedures in the Oracle VM do not allow standard JDBC connections, however, a special direct internal JDBC driver can be used from within the database when a database connection is needed from the internal Java stored procedure.

B. Implementing the new Operators

When a new publication arrives at a broker it is compared to the set of subscriptions stored in the broker to check if it should be forwarded to a subscriber. Standard non-temporal subscriptions contain a filter, which is made up of a set of *operator/value* pairs to be applied to named attributes. If the newly arrived publication contains attributes with the same names, and the values compare successfully with the value given in the subscription filter according to the given operator, then that subscription is matched. Temporal subscriptions extend standard subscriptions. They can contain the set of *standard-operator/value* pairs to be applied to named attributes, but, they also contain at least one *temporal-operator/inner-filter* pair. Here, the passed inner-filter is used to retrieve from the data store a set of historical publications that match the inner-filter. (The passed inner-filter can itself be a temporal filter, which then must contain its own *inner*-inner-filter). Given the set of historical publications returned, the time-based characteristics of those older publications are then compared to the newly arrived publication in a way that is appropriate for the temporal operator.

Each of the three new operators were individually designed and implemented with a method called apply() which performs the operation. The apply() method first selects the appropriate data store and invokes hasMatchingPublication() to retrieve the set of historical publications that have matched the standard content-/semantic-based filter (inner-filter). A limit can also be specified for the AFTER and FILTER operators to limit the size of the historical time window within which the data store should search for past publications. The retrieved publications are then compared to the newly arrived publication. It should be noted that the size of the temporal window can be set on a filter-by-filter basis, even with different window sizes in different filters in the same subscription. This fine-grained and flexible approach ensures that the subscriber can specify detailed subscriptions for only the events of interest.

Two conditions must hold for the AFTER operator to match. The *end* time, Y_E, of the stored event Y must be *less than* the value of the reference attribute X_R, (i.e., $Y_E < X_R$), and, the end time must be *greater than or equal* to the value of the reference attribute less the limit, L (i.e., $Y_E \geq (X_R - L)$).

For the DURING operator, two different conditions must hold. The *start* time, Y_S, of the stored event must be *less than or equal* to the value of the reference attribute (i.e., $Y_S \leq X_R$), and, the *end* time of the stored event must either be *greater than or equal* to the value of the reference attribute or it must be null (i.e., $Y_E \geq X_R \vee Y_E$ is null).

The FILTER operator, described earlier, is somewhat different. It can be used to compare the value of an arbitrary named attribute (not just start or end time) of the stored publication to the value of the same name specified in a newly arrived publication, using an arbitrary content-/semantic-based operator. Therefore, the FILTER operator requires the name of the attribute to be used and the operator to use in the comparison.

In order for our implementation to correctly and efficiently function within the Siena/KBN framework, we need to define when a filter can be aggregated by a more general filter. This subscription aggregation relationship, called covering in Siena, defines how subscriptions can be efficiently stored and searched in an individual broker, and defines which subscriptions are passed around the network to neighbouring brokers. (Only the most general subscriptions are distributed to neighbouring brokers, as these will suffice to capture the requirements of all of a broker's subscriptions). For the operators in our extension, the covering relationships depend heavily on the internal target filters used in the operators, and cannot be determined otherwise. If two filtering constraints contain the same temporal operator, and the internal target filter of the operator in the first covers that of the second operator, then the first filtering constraint covers the second.

VI. Motivating Scenarios

A. First Deployment Scenario

Many managed systems rely heavily on logging, whether of alerts, warnings or errors, where log messages are based on the system's current state and context. To be useful, they must be very fine grained, but this challenges the system administrator who must monitor and search the logs to find the messages that are useful to detect and fix errors and warnings. Most systems filter log messages, based on their logging level (e.g., level 5 = ERROR, level 1 = TRACE), where important messages are kept and occurrences of log message with lower importance are ignored.

Consider a case where only highest level messages are

displayed to an administrator, while others are ignored (< WARNING). However, if there may be a fault in the near future or if there is some other particular pattern of messages that may require more attention (such as a potential intrusion pattern), the administrator may want to lower his logging filter match. For example, if three WARNING messages, each containing a value "login failed", are detected less than two seconds AFTER each other, then the logging level should be immediately dropped to INFO for a short time so that more information, such as the location of the warning can be logged.

B. Second Deployment Scenario

While numerous works describe the benefits of using semantic mark-up in the area of network fault management (e.g. [14]), and so are suited to the knowledge-based networking approach, there has been little previous progress in providing infrastructure support to capture causal and temporal dependencies between events. Aspects of the causal relationship between types of events can be easily captured in a causal ontology, and thereby subscribed for using the generic ONTPROP operator (see [10]), however, without temporal relationships it does not check if any fault/warning which could have caused a matched fault actually happened.

For example, consider a scenario where an end-to-end virtual private network (VPN) connection is supported by a two logical switched paths using a core routing protocol (e.g. Border Gateway Protocol (BGP)) and a data carrying/switching protocol (e.g. Multiprotocol Label Switching (MPLS)). An Simple Network Management Protocol (SNMP) "ifdown" *trap* can signal that a device interface has failed, thereby causing a "linkdown" trap, which in turn indirectly causes "mplsTunnelDown" and "mplsLdpSessionDown" events, which themselves later cause the VPN connection to fail. The individual device/port which caused this failure can be easily detected by subscribing to any low level fault known to indirectly cause "mplsTunnelDown" faults, anytime in the 10 seconds BEFORE notification that the VPN connection has failed.

VII. Evaluation

The first series of tests attempt to establish how the processing time for the extension grows with respect to the number of stored publications, starting with an empty data store. For this test, we connected an example publisher and an example subscriber to a single temporally-extended KBN broker. The example subscribing client generated fifty subscriber entities, where each subscribed with one of the subscriptions shown in Table 3, randomly-selected with a uniform description (a filter used as a target value is denoted by F_X where X is the filter name). Note, filter B in Table 3 uses the W3C Wine Ontology [26]. The filter constraint ("B" $ONTPROP_{<wine:hasColor>}$ <wine:red>) will successfully match a publication that has an attribute called "B" containing an ontological individual that uses the object property "wine:hasColor" with the value of "wine:red".

Filter	Attribute	Operator	Target
A	A	<	8
B	B	$ONTPROP_{<wine:hasColor>}$	<wine:Red>
C	A	$FILTER_{<}(200)$	F_B
D	*End*	*DURING*	F_A
E	*Start*	*AFTER*(500)	F_A
F	*Start*	*AFTER*(125)	F_A
X[a]	C	>	10
G	A	$FILTER_{<}(200)$	F_X

[a] *This filter is only used as the internal filter for the following filter operator.*

Table 3: Special temporal attributes

The publisher published 250 events, one beginning every 10 seconds and ending between three and eight seconds later (to test varying event durations). The publications contained two attributes, A and B, each of which were assigned a random value. A was set to an integer between one and ten, inclusive, while B was set to the identifier URL of one of ten wines from the Wine Ontology, five red and five white.

These tests were run for each of the three data store implementation described above, (MySQL, Simple Oracle, and Advanced Oracle). Based on the algorithms employed, we predicted that the processing time would scale at most linearly with the number of events stored in the data store. The results of the tests using the simple Oracle and MySQL data stores implementations are shown in Figure 2. As the graph indicates, the average processing time for each publication does appear to increase linearly with the number of events stored in the data store, at least for the MySQL results which match a linear regression with a high degree of confidence (R=0:989). The simple Oracle JDBC implementation strongly fits a linear regression (R=0:854), but is also a strong fit for a logarithmic regression based on the test data (R=0:819), showing that the processing time for the Simple-Oracle store may grow with the log of the stored events rather than growing linearly.

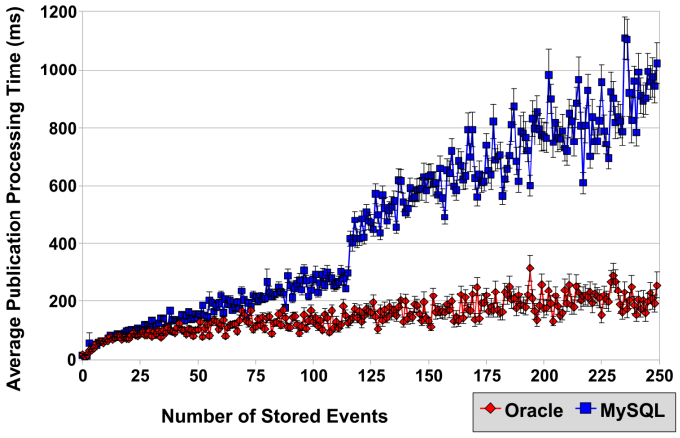


Figure 2: Comparing the MySQL and Simple Oracle data stores

The more advanced Oracle store class also displayed a linear increase with the number of stored events (R=0:999), but the increase was dramatically greater than the simpler

Oracle store, which did not utilize the Java stored procedure invocations used by the more complex version. Figure 3 compares the two Oracle data stores directly based on their results from Test 1. As shown, the advanced Oracle store grows linearly, but for only 250 stored events the processing time for each publication takes an average of nearly seven seconds. This unacceptably long delay is likely due to the overhead of invoking the Java stored procedure repeatedly, especially since the code base size and complexity of the procedure is quite large, as discussed in section IV:A:3 above. This behaviour was not affected by subsequent changes in the configurable variables of the test (e.g., delay between publications, number of publications). It is likely that the matching function invoked by the stored procedure is more complex than this feature was intended to perform.

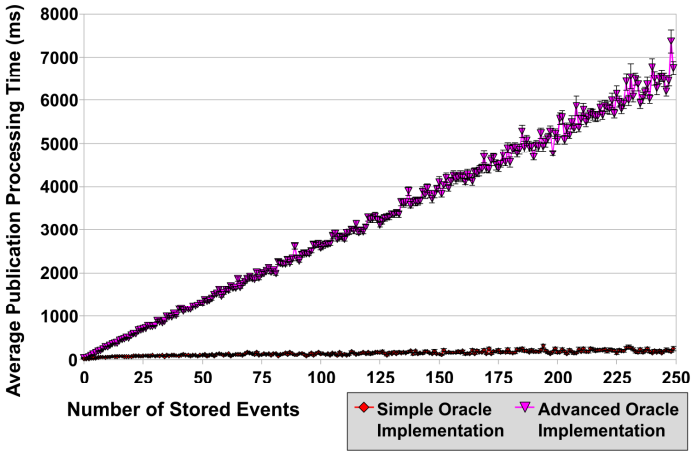


Figure 3: Comparing the Simple and Advanced Oracle data stores

A second test utilized much the same set-up as the first, involving a single temporally-extended KBN broker, a simple subscriber, and a simple publisher. However, the second set of tests operated on a pre-filled database containing 50,000 randomly-generated events (using the same publication attributes as the first test) and focused specifically on the time taken to process the FILTER operator, which we determined takes the most time due to its unconstrained nature (the temporal operators took an average of around 10ms to 63ms to process). The fifty subscribers for the second test used one of two filters, randomly selected from those described in Table 4. The publisher used for the second test is the same as the publisher used for the first test, but with a three second delay.

Filter	Attribute	Operator	Target
X[a]	B	$ONTPROP_{<wine:hasColor>}$	<wine:Red>
A	A	$FILTER_{<}(\alpha)$ [b]	F_X
Y[a]	C	>	10
G	A	$FILTER_{<}(\alpha)$ [b]	F_X

[a] *This filter is only used as the internal filter for the following filter operator.*

[b] *The limit on the filter operators used in these subscriptions is one of the variables manipulated in this test.*

Table 4: Subscriptions for Test 2

The second set of tests were each run with varying values for the filter operator limit (200, 500, 1000, 2000, and 5000 results), all using the Simple Oracle implementation. We expected that the average processing time would remain nearly constant, due to the limit imposed on the filter operator result set size. As expected, the average publication processing time for the second test (with a filter operator limit of 200 results) using the simple Oracle store hovered around 200 milliseconds, as shown in Figure 4.

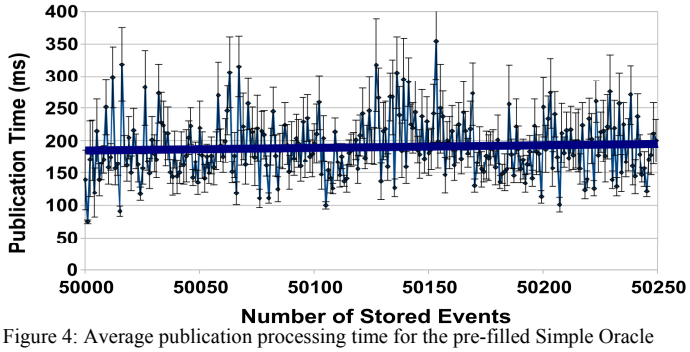


Figure 4: Average publication processing time for the pre-filled Simple Oracle data store, using the FILTER operator with a filter limit of 200.

These results show that the processing time scales linearly not in the number of total events stored in the database, but rather that the processing time is a function of the size of the result set returned for each query. We expected that the publication processing time will increase approximately linearly as the limit on the result set increases. In order to test this prediction, we ran the second test ten times each with filter operator limits of 200, 500, 1000, 2000, and 5000 results. The average publication processing time for each limit (along with standard error) are shown in Figure 5. As shown, the publication processing time scales almost linearly, with a small (super-linear) increase in processing time as resources become scarce with larger filter operator limits.

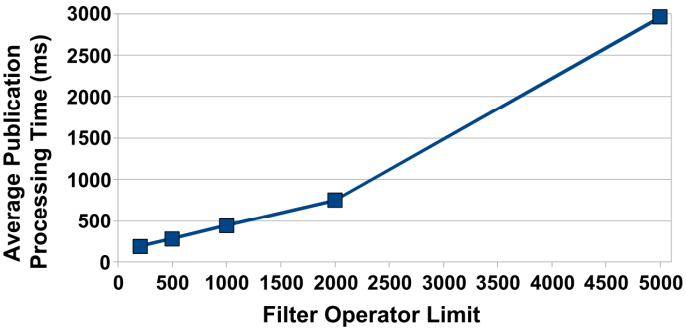


Figure 5: Average publication processing time based on the FILTER operator's filter limit.

VIII. Discussion and Further Work

This paper presents a design, implementation, evaluation and two motivating deployment scenarios to demonstrate a new CEP approach for composite event detection − that of detecting composite events using predicate matching using historical event data. Our results demonstrate that this approach can be implemented in a scalable manner and our scenarios motivate the more expressive subscriptions supported. It is particularly envisioned that the collection,

aggregation and correlation of network faults and events will be made easier, further enhancing the usefulness of Knowledge-based Networking in this domain [13][16].

While this work has increased the expressiveness of the subscriptions in the underlying KBN without sacrificing scalability, the temporal operators model only a subset of Allen's interval algebra, and do not yet allow for uncertain time intervals. Further work will also introduce logical composition extensions to the new temporal reasoning support, e.g. conjunction (where two or more events must be detected in any order) and disjunction (where any one of two or more events can be detected). These additional operators could easily achieved using the same predicate-matching approach used here.

ACKNOWLEDGEMENT

This work was partially funded by the Irish Government as part of the SFI Strategic Research Cluster ("FAME"). Grant No 08/SRC/I1403

BIBLIOGRAPHY

[1] Network Systems Architects, Inc. "SMARTS InCharge Application Services Manager." Retr. Sept 2009 at: http://www.nsai.net/products/incharge-asm.shtml.

[2] Hewlett-Packard Development Company. "Looking for HP OpenView?" Retr. Sept 2009 at: http://openview.hp.com

[3] P.T. Eugster, P.A. Felber, R. Guerraoui, A-M. Kermarrec. "The many faces of publish/subscribe." ACM Computing Surveys, 35(2), June 2003.

[4] International Business Machines (IBM). "The Gryphon project." Retr. Sept 2009 at: http://www.research.ibm.com/distributedmessaging/gryphon.html.

[5] A. Carzaniga, D.S. Rosenblum, A.L. Wolf. "Design and evaluation of a wide-area event notification service." ACM Trans. Comput. Syst., 19(3), 2001.

[6] P. Pietzuch, J. Bacon. "Hermes: A distributed event-based middleware architecture". Proc. of the International Conference on Distributed Computing Systems (ICDCSW '02), Washington, DC, USA, 2002.

[7] A. Carzaniga, A.L. Wolf. "Forwarding in a content-based network." Proc. of the conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '03), New York, NY, USA, 2003.

[8] R. Baldoni, R. Beraldi, L. Querzoni, A. Virgillito. "Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA." The Computer Journal, 50(4), 2007

[9] Z. Jerzak, C. Fetzer. "Bloom Filter based routing for content-based publish/subscribe". Proc. of the international conference on Distributed event-based systems (DEBS '08), Rome, Italy, 2008.

[10] J. Keeney, D. Roblek, D. Jones, D. Lewis, D. O'Sullivan. "Extending Siena to support more expressive and flexible subscriptions." Proc. of the international conference on Distributed event-based systems (DEBS '08), Rome, Italy, 2008.

[11] M. Cilia, C. Bornhvd, A. P. Buchmann. "Cream: An infrastructure for distributed, heterogeneous event-based applications." Proc. of the International Conference on Cooperative Information Systems (COOPIS '03), Catania, Sicily, Italy, 2003.

[12] J. Wang, B. Jin, J. Li. "An ontology-based publish / subscribe system." Proc. of the international conference on Middleware (Middleware '04), New York, NY, USA, 2004.

[13] J. Keeney, D. Lewis, D. O'Sullivan. "Ontological semantics for distributing contextual knowledge in highly distributed autonomic systems." J. Netw. Syst. Manage., 15(1), 2007.

[14] J, Keeney, D. Lewis, D. O'Sullivan, A. Roelens, A. Boran, R. Richardson. "Runtime Semantic Interoperability for Gathering Ontology-based Network Context." Proc of the Network Operations and Management Symposium (NOMS '06), Vancouver, Canada, 2006.

[15] D. Lewis, J. Keeney, D. O'Sullivan, S. Guo. "Towards a Managed Extensible Control Plane for Knowledge-Based Networking", Proc. of the International Workshop on Distributed Systems: Operations and Management (DSOM 2006), at Manweek 2006, Dublin, Ireland, 2006.

[16] T. Wei, D. O'Sullivan, J. Keeney. "Distributed Fault Correlation Scheme using a Semantic Publish/Subscribe system." Proc. of Network Operations and Management Symposium (NOMS 2008), Salvador, Bahia, Brazil, 2008.

[17] E. Wu, Y. Diao, S. Rizvi. "High-performance complex event processing over streams." Proc. of the SIGMOD international conference on management of data New York, NY, USA, 2006.

[18] S. Gatziu and K.R. Dittrich. "Detecting composite events in active database systems using petri nets". Proc. Of the International Workshop on Research Issues in Data Engineering: Active Database Systems, Houston, Texas, 1994.

[19] N.H. Gehani, H.V. Jagadish. "Ode as an active database: Constraints and triggers." Proc of the International Conference on Very Large Data Bases (VLDB '91), San Francisco, CA, USA, 1991.

[20] Middleware Systems Research Group. "PADRES: A reliable publish/subscribe middleware." Retr. Sept 2009 at: http://research.msrg.utoronto.ca/Padres.

[21] R.E. Gruber, B. Krishnamurthy, E. Panagos. "The architecture of the ready event notification service." Proc. Of the Workshop on Electronic Commerce and Web-Based Applications, at ICDCS '99, Austin, TX, USA, 1999.

[22] J.F. Allen. "Towards a general theory of action and time." Artificial Intelligence, 23(2), 1984.

[23] Hewlett-Packard Development Company. "Jena - A semantic web framework for Java". Retr. Sept 2009 at: http://jena.sourceforge.net.

[24] C. Murray. "Oracle Database Semantic Technologies Developer's Guide, 11g Release 1 (11.1) (B28397-05), July 2009.

[25] T. Das, S. Maring, R.Sapir, M. Wiesenberg. "Oracle Database Java Developer's Guide", 11g Release 1 (11.1) (B31225-03), September 2007.

[26] World Wide Web Consortium (W3C). "Wine ontology". Retr. Sept 2009 at: http://www.w3.org/TR/owl-guide/wine.rdf.