High-Level Constructs in the READY Event Notification System

Robert E. Gruber, Balachander Krishnamurthy and **Euthimios Panagos**
{gruber, bala, thimios}@research.att.com
AT&T Labs — Research

1 Introduction

There is growing interest in the use of general-purpose event notification services as "middleware" for gluing together independently-developed distributed applications. This approach was first discussed by Oki *et.al.*in 1993 [14], where an early version of the (now) TIBCO Information Bus [20] was described. Since then, many other commercial products have been developed that accept events from event suppliers (publishers) and deliver them to event consumers (subscribers). Publish-subscribe mechanisms have also been retro-fitted to existing middleware products, including persistent message queue products and TP monitors. There is also standards work in this area, including the CORBA Event Service [15] and more recently the CORBA Notification Service [16], a feature-enriched version of the Event Service.

Despite all of these products and the standardization efforts, and despite the fact that event-driven computation is not a new idea, there has been relatively little work on high-level constructs for event services. The most basic event service has four kinds of entities (supplier, consumer, event, event service) and three basic functions (supply an event to the service, register interest in a kind of event, unregister interest). Existing commercial event services do not provide event models that are much richer than this most basic service. For example, there are typically no constructs that enable operations over multiple entities.

This paper describes high-level constructs (operating over multiple events, multiple consumers, *etc.*) provided by READY, an event notification system being developed at AT&T Labs [7]. Entities in the READY model include: consumer specifications that match over both single and compound event patterns; communication sessions that manage quality of service (QoS) and ordering properties for event delivery; grouping constructs for both specifications and sessions; event domains and boundary routers that bound the scope of event distribution and control the mapping of events across domains. The result is a more powerful and flexible event service.

The remainder of the paper is organized as follows. Section 2 offers an overview of related systems. In Section 3, we focus on client interactions with READY. Section 4 introduces READY event types. Section 5 describes specifications, which control the actions performed by READY servers due to supplied events. Section 6 describes constructs for managing boundaries between event domains. Finally, Section 7 discusses the current status of READY as well as some general issues raised by our work.

We are in the process of completing an initial implementation of READY. At the same time we are exploring the merits of READY's high-level constructs by using READY as a core component of a unified messaging system [2]. Some of the examples presented here are taken from that domain.

2 Related Work

The CORBA Notification Service [16] offers a decoupled, publish-subscribe model of communication via *notification channels*. This service significantly enhances the CORBA Event Service [15] by adding event filtering, durable connections, and delivery-guarantee semantics. Suppliers supply events to notification channels while consumers register simple filters with channels. A Notification channel passes to each registered consumer all supplied events that pass its filters. Suppliers and consumers do not interact directly with a channel, but rather with channel proxies. By attaching a set of consumers to a single proxy object, one can arrange to have these consumers share a set of filters, thus providing a limited form of READY's consumer session groups. In addition to better grouping mechanisms, READY specifications are much more powerful than simple filters, providing compound matching, aggregation, local actions, and so on.

READY also masks distribution issues: READY is implemented as a set of cooperating servers that together implement all necessary event routing, where suppliers and consumers can simply connect to the nearest server. (There is currently an RFP for managing a distributed topology of CORBA Notification Channels, but at present such a topology must be put explicitly constructed by clients of the Notification Service.)

Microsoft's Distributed Component Object Model (DCOM) extends the Component Object Model (COM) [3] by adding network-based component interaction. COM provides event notification capabilities via specialized objects, referred to as "connectable objects," which establish connections between event suppliers and event consumers. However, COM does not support a decoupled mode of communication nor does it provide any of the higher-level constructs provided by READY.

NEONet [13] is a persistent messaging middleware offering a publish-subscribe mechanism with format parsing and conversion capabilities. Content-based subscription rules are supported, *i.e.,* multi-field matching for single events. Similar to READY, subscribers in NEONet can specify message formats so that they receive the same format from multiple suppliers; this is done by selecting some fields from the original message and adding additional data. Unlike READY, NEONet does not offer local action support, grouping constructs, and compound event matching functionality.

Vitria's Velocity server [18] and Tibco's Information Bus (TIB) [20] are two widely used publish-subscribe middleware products. Velocity, which is based on the CORBA event service, uses multiple event channels for event notification, offers access control lists for security, caches and replicates events for increased performance and provides quality of service guarantees. TIB, which is based on a subject-based addressing scheme, supports similar functionality. In contrast to READY, these products do not offer grouping constructs, multi-event matching, local actions and event aggregation.

Level8 Systems EventWorks [19] is a reliable and scalable publish-subscribe middleware product which provides rule-based event processing. EventWorks, based on the CORBA event service, uses the concept of themes, which is similar to event channels. Themes use a hierarchical name space that is stored in a directory. Attributes may be associated with themes, and they can be used for filtering. The only similarities between READY and EventWorks is the use of self-describing events and the support for quality of service levels that provide total ordering of events, duplicate free notifications and replication for redundancy.

Yeast [11] is a general purpose event notification system that provides a global space of events that is visible and shared in a wide area network. A event action specification can have a pattern of events which, when matched, triggers a specified action. High-level applications are built as collections of specifications. READY extends the Yeast specification language by adding compound matching of events, quality of service directives and several other language constructs.

The OPERA group at the University of Cambridge has ongoing work on event-based distributed computation [1, 9, 12]. This work includes a compound matching service that has constructs similar to READY's compound matching operators. In terms of functionality, one of the main differences between this project and READY is our study of higher-level constructs such as the session and grouping constructs. (There are also differences in the projects in terms of research focus w.r.t. implementation issues.)

TAO [8] is an object-oriented, real-time event service. One of the main contributions of TAO is the extension of the CORBA event service to suit real-time systems and support periodic, rate-based event processing and efficient event filtering and correlation. TAO provides source-based and type-based event filtering, event correlations, and real-time event dispatching. Similar to READY, TAO suppliers specify the types of events they generate, and consumers register filters and quality of service demands, with real-time constraints being the most important. In contrast to the rich filtering capabilities provided by READY, TAO only offers source-based, type-based, source- and type-based filtering, and simple event correlation based on conjunctions and disjunctions of events.

Active databases offer an *event-condition-action* (ECA) to support database functionality such as view materialization, index maintenance and integrity constraint enforcement [21]. The ECA mechanism consists of an event expression, one or more conditions and an action that is triggered when the conditions are satisfied upon event occurrences. Active databases support composite events for event correlation. However, virtually all active databases are based on a centralized architecture and, therefore, are not suitable for distributed systems. In addition, with the exception of HiPAC [4], the actions specified as part of the ECA rules are internal to the database and do not interact with external applications.

3 Client-READY Interactions

READY clients interact with READY using sessions. Sessions can be *admin sessions, supplier sessions* or *consumer sessions*. Admin sessions are used for creating/destroying supplier and consumer sessions and session groups and adding/removing sessions to/from session groups. They are also used for supporting additional operations that are not specific to a particular supplier or consumer session.

Supplier sessions are used to supply events to the service. Before a supplier session S can be used for this purpose, the supplier must register a description of the kinds of events that it is going to supply using S. READY is free to discard events supplied to session S that do not fit the description. The normal case is to register the event types that will be supplied. Additional constraints can be registered, *e.g.,* a supplier could indicate that the priority field for all events it supplies will never be greater than 10.

Consumer sessions are used for: registering/removing specifications (discussed in the next section), changing the status of registered specifications, creating/destroying groups of specifications, adding/removing specification to/from groups, and setting delivery properties for notifications from these specifications. The suspend and resume operations are used to change the status of a registered specification, *i.e.,* disable or enable matching over the specification, respectively. Specifications can be grouped together into named groups to facilitate rapid changes to sets of specifications (*e.g.,* when suspend is used on a group, matching is disabled over all specifications that belong to the group).

Important delivery properties include quality of service (QoS) and ordering constraints. (Events that reach consumer within some time window are ordered with respect to each other according to ordering constraints.) Two examples of delivery/ordering QoS combinations: best-effort delivery ordered by timestamp; reliable delivery ordered by priority.

A consumer session can be *disconnected* from the event service. Conceptually, the service is responsible for retaining notifications until the session is *reconnected*, at which point notifications are delivered. A combination of factors, such as the QoS specification for the consumer session and time-to-live fields in events, determines how long the service actually retains notifications for a disconnected session.

The operations described above for individual consumer or supplier sessions can also be applied to session groups. When a specification (or specification group) is associated with a consumer group, notifications from this specification go to all sessions in the consumer group. The delivery properties of all members of a session can be set via the group. Although not (yet) part of the READY design, it seems clear that concepts from group communication systems [17, 5] can be applied to session groups. For example, one could optionally specify that all consumer sessions in a session group must receive notifications in a consistent order.

4 Event Types

This section describes READY event types and event encoding issues, where the two key properties are extensibility and self-describing encoding. Extensibility is accomplished in two ways: subtypes describe extensions that are used uniformly across many events, while arbitrary additional fields can be added to individual event instances.

In READY, an event type definition specifies a set of required and optional fields, where each field contains a field name, type identifier, and value. The field type specifies the encoding of the value. Field types include scalar values, such as string, real, integer, set types, and event types. Event fields are self-describing: the name and type of a field can be determined from the field itself. READY types can have subtypes. A subtype declaration simply adds additional required or optional field specifications to those of its parent type.

If event type E has required field R of type T, all E events must have an R field with type T. If event type E has an optional field O of type T, an E event does not need to have an O field; however, if O *does* appear in an E event, it must be of type T. Any event encoding, regardless of its type, can have *extra fields*. Access to such fields always requires run-time type checking.

An event encoding must include a field with name EType whose value identifies the type of the event. READY can be configured to assumes a correct encoding of the required fields for this event type immediately follows the EType field. It can also be configured to parse and perform sanity checks on each field. Event type identifiers are conceptually encoded as strings, but the encoding can be optimized.

Compound type identifiers such as Mail.Msg.New can be used. For our messaging framework, we use the convention that the structure of type identifiers matches the type hierarchy. For example, Mail.Msg is a subtype of type Mail, and Mail.Msg.New is a subtype of Mail.Msg. This convention simplifies integratation of a READY-based system with a publish-subscribe system: simple matching expressions over a compound EType mimic the matching over compound subject names found in many publish-subscribe systems.

5 Specifications

A *specification* is composed of one or more event *matching expressions* together with *actions* that are triggered by successful matches. A specification is always associated with a consumer session (or session group), and most actions implicitly operate over entities associated with this session (group).

Matching expressions bind event variables to events that match a specified event pattern. A simple matching expression is a pattern that matches a single event. Such expressions are also called event filters. A compound matching expression is a pattern that describes a combination of events. Such a pattern often corresponds to a high-level event, *i.e.,* an event that subsumes the set of lower-level events that matched the pattern. Thus, compound matching adds an abstraction mechanism to the event service.

Actions are imperative statements that are executed at specified points during the matching of a specification. An assignment action binds an event variable to a newly-created event. This is most often used to perform *aggregation*, where a set of matching events is summarized by a single new event. Actions that operate over event variables include the notify action, which causes delivery of one or more events to the associated consumer session (group), and the $\mathrm{announce}$ action, which acts as a supplier of a new event. Announce is the only action that has global impact rather than acting on the associated session (group).

Actions that perform operations over the state of the associated session (group) are referred to as *local actions* since executing these actions only involves changing the internal state of the READY servers; they do not require contacting suppliers or consumers. All of the operations over specifications, specification groups, sessions, and session groups that are described in the previous section are available as local actions.

5.1 Matching Expressions

Matching expressions bind event variables to events that match a specified event pattern. For example,

$$\{\mathrm{Ev1:\ MyAlarm} \mid \$\mathrm{Priority} > 5\}$$

binds Ev1 to the first MyAlarm event with Priority greater than 5. This expression can be read as "find an Ev1 with type MyAlarm such that Ev1's Priority field has value greater than 5."

The general form for an expression that matches a single event is:

{ *event_var* : *event_type* | *simple_expression* }

One can omit the colon and *event_type* or the *simple_expression*, but not both. One can also omit the variable; an expression with no variable is referred to as an anonymous matching expression. Since no event variable is bound, the state of an event that matches an anonymous matching expression is not available after the match succeeds. Since the type of an event is encoded in an event header field, specifying an event type with an event variable is simply a shorthand:

{Ev1: T | *simple_expression* }

is equivalent to

{Ev1 | \$EType == 'T' and
simple_expression }

The *simple_expression* is any legal expression from the event filter grammar specified for the OMG Notification Service [16]. The \$ symbol stands for the event being matched against, and component expressions such as \$Priority evaluate to values of components of this event. One can write component expressions that select sub-(sub- ...) parts of top-level event fields. See the Notification Service document for a complete description of the filtering grammar. For READY we extended this grammar by allowing components of event variables to be accessed using components that begin with :*event_var*: rather than \$. Thus, we can re-write the example above as:

$$\{\mathrm{Ev1:\ MyAlarm} \mid \mathrm{:Ev1:Priority} > 5\}$$

Simple matching expressions can be combined to form compound matching expressions using operators '&&' (and), '||' (or), ';' (then), and not. These compound matching operators are based on the language used for Yeast [11], an event-action service developed by AT&T Labs researchers in 1994-95. The same operators have since been used in many other contexts, as discussed in the related work section.

For example,

{Ev1: MyAlarm | :Ev1:Priority > 5} &&
{Ev2: MyAlarm | :Ev2:Priority == :Ev1:Priority}

binds Ev1 and Ev2 to the first pair of MyAlarm events that have an equal Priority value that is greater than 5. As you would expect, the '&&' and '||' operators begin matching both LHS and RHS at the same time, where '&&' matches successfully once both sides match, while '||' matches successfully once either side matches. In contrast, the sequencing (then) operator, written as a semi-colon, does not attempt to match the RHS until the LHS has matched. Thus

{Ev1: MyAlarm | :Ev1:Priority > 5} ;
{Ev2: MyAlarm | :Ev2:Priority > :Ev1:Priority}

binds Ev1 to the first MyAlarm event with Priority greater than 5, and *then* binds Ev2 to the first MyAlarm event that follows Ev1 with Priority greater than Ev1's Priority. The not operator is always paired with a sequencing operator:

not *expr-Bad* ; *expr-Good*

tries to match both expressions and succeeds if *expr-Good* successfully matches before a *expr-Bad* match occurs. A common use is of not is in expressions of the form

expr-A ; not *expr-Bad* ; *expr-B*

This expression is equivalent to

expr-A ; *expr-B*

with the additional constraint that *expr-Bad* must not successfully match between the successful match of *expr-A* and the successful match *expr-B*.

Another form of compound matching is to specify a sequence of events which all match the same "element pattern" (a matching expression used for each element of the sequence). The general form for a sequence matching expression is:

{ *event_var*[j..k] : *event_type* | *simple_expr* }

The *event_var*[j..k] indicates that the event variable will be bound to a sequence of events, where j and k indicate the minimum and maximum number of events in this sequence, respectively. j can be zero, indicating no lower bound. k can be an asterisk, indicating no upper bound.

As a top-level expression, sequence matching completes when the minimum number of successful element matches occur, thus expressions using Ev1[3..3] or Ev1[3..5] or Ev1[3..*] all result in a sequence of length 3 if used at the top level. When contained within a larger expression, however, a sequence match is *successful* once the minimum number of element matches occur, but elements continue to be added to the sequence until either the maximum number of element matches occurs or a containing expression successfully matches. For example,

```
{Ev1[3..5]: MyAlarm | $Priority < 20} ;
{Ev2: MyAlarm | $Priority >= 20} ;
```

This specification matches a sequence of between three and five MyAlarm events with Priority less than 20 followed by a MyAlarm event with Priority 20 or greater. The Ev1 match is successful as soon as three events with Priority less than 20 are found; at this point, as soon as the Ev2 match occurs, the entire expression matches. While waiting for Ev2 to match, up to two additional Ev1 elements are added to the Ev1 sequence (if further events with Priority less than 20 occur before an event with Priority greater than 20).

5.2 Action Expressions

Action expressions are imperative actions operating over the state of currently bound event variables or over the state of entities associated with the specification's consumer session or session group. An action expression can only occur as the LHS or RHS of the sequencing operator. When the LHS of the sequencing operator is a matching expression, the RHS action is executed only after the match succeeds. When the LHS is an action, this action is executed before the RHS is evaluated (the RHS could be either an action or matching expression).

The most common action is the notify action:

```
{Ev1: MyAlarm | :Ev1:Priority > 5} ; notify Ev1
```

Once Ev1 is bound to an event, the consumer is notified of this event.

In the original READY design, actions could only occur as the final expression in a specification. However, relaxing this rule proved to be useful. Note the difference between the following two specifications:

```
{Ev1: MyAlarm | :Ev1:Priority > 5} ;
notify Ev1 ;
{Ev2: MyAlarm | :Ev2:Priority > :Ev1:Priority} ;
notify Ev2
```

and

```
{Ev1: MyAlarm | :Ev1:Priority > 5} ;
{Ev2: MyAlarm | :Ev2:Priority > :Ev1:Priority} ;
notify Ev1,Ev2
```

For the first specification, the consumer is notified when the first event with Priority greater than 5 is found; if an event with an even higher priority occurs after this event, the consumer is notified of this second event. In contrast, for the second specification both events must be found for a notification to occur.

When a consumer callback function is called (to handle a notification) it is given a sequence of events as input. If this is a notification for just one event, the sequence has length one. For the second specification above, the sequence would have length two: a single notification would deliver both Ev1 and Ev2. To cause separate notifications with one event per notification, one would write:

matching expressions ; notify Ev1 ; notify Ev2

For a successful sequence match, actions can select some or all of the matched sequence elements. Here is the sequence match example from above with an added notify action:

```
{Ev1[3..5]: MyAlarm | $Priority < 20} ;
{Ev2: MyAlarm | $Priority >= 20} ;
notify Ev1[0..#],Ev2
```

For sequence variables such as Ev1, the first element is always at index zero (sequences are zero based) while '#' refers to the index of the last element. Event selection expressions such as Ev1[0..#] can be used to specify a sequence or set of events. This particular expression selects all of the events in Ev1. In a context where an event set or sequence is expected, Ev1 by itself can be used as a shorthand for Ev1[0..#], thus the example above could simply have used notify Ev1,Ev2. In general, a comma-separated list of indices and index ranges can be used. Examples: Ev1[0,2..#] selects the first element and all events from the third element to the last; Ev1[0,#] selects the first and last elements; Ev1[1..(#-1)] selects all elements except the first and last.

Selecting only a subset of all matching events is one way to reduce the size of notifications. In addition, it is often useful to define a new event which summarizes the state contained in several matching events. READY supports such *event aggregation* via event definition actions. Here is an example from our unified messaging framework:

```
{Ev1, Ev2: Mail.Msg.New |
    :Ev1:From == :Ev2:From and
    :Ev2:Subj ~ :Ev1:Subj} ;
Ev3 = new Mail.DupSubj
    ( Id1 = :Ev1:MsgId,
      Id2 = :Ev2:MsgId ) ;
notify Ev3
```

This specification matches a pair of Mail.Msg.New events describing new mail messages such that the same user sent both messages and one subject string contains (is a substring of) the other. When matching events are found, a new event Ev3 is created and delivered to the consumer as a notification. Note that Ev3 contains only the message identifiers of the matching mail messages, while the two Msg.New messages contain fields describing all of the useful mail header fields (and optionally a message body, for mail messages with short bodies). A notification containing Ev3 is significantly smaller than a notification containing the sequence Ev1,Ev2.

While an event can be created "from scratch" via a parenthesized set of assignments to fields, as shown above, READY also supports shorthands which simplify event aggregation. When considering support for aggregation, we observed the following: events contain named fields; the set of events that match a compound specification is a set of sets of named fields; aggregation often involves flattening the set of sets back to a simple set of named fields (where fields need to be removed, kept as-is, or combined to produce new fields). Since two events can have a field with the same name, there must be a way to rename a field at the time flattening occurs, *e.g.,* so that both fields can be retained as-is. It is not hard to invent set operators that select or remove named fields, rename fields, and so forth; this is what we did. The above aggregation can be written as:

```
Ev3 = new Mail.DupSubj
    (Ev1|(MsgId->Id1), Ev2|(MsgId->Id2))
```

5.3 Advanced Features

The form

matching_expr where *where_expr*

attaches a *where expression* to a matching expression, further constraining the set of possible matches. The where expression must be a boolean expression over the values of the event variables introduced in the associated matching expression (where cannot trigger additional event matching). This form is useful for expressing inter-event constraints, especially constraints over sets of events. For example, here is an expanded version of the duplicate subject detector given above:

```
{Ev1, Ev2: Mail.Msg.New |
    :Ev1:From == :Ev2:From and
    :Ev2:Subj ~ :Ev1:Subj}
    where span_le(Ev1,Ev2; $StartTime; 1h) ;
Ev3 = new Mail.DupSubj
    (Ev1|(MsgId->Id1), Ev2|(MsgId->Id2)) ;
notify Ev3
```

The where expression constrains the matching to a pair of events that occur within one hour of each other. Such a time-limited specification is far preferable to the previous version, which is open-ended. Open-ended specifications can cause an arbitrary amount of intermediate matching state to be generated at READY servers (and never discarded). For this reason, READY imposes a default timespan constraint on open-ended specifications. This default can be explicitly over-ridden (with either shorter and longer timespans) using where expressions such as the one above.

The span_lt and span_le operators are only valid in a where expression. Each take three arguments, separated by semi-colons. The first argument is a set of events. The second argument is an expression from the simple expression grammar which extracts a numeric or timestamp value; this expression is used to map the set of events to a set of values. ($ stands for each event to be mapped.) The third argument is an upper bound on the span between the minimum and maximum of the resulting values. span_lt requires the span to be less than this bound, span_le also allows the span to equal this bound. In the example above, the StartTime field is extracted from Ev1 and Ev2 and the distance between the two times must be less than or equal to 1 minute. (StartTime is the time an event becomes valid. For a new mail event it would be the time that the event supplier, such as a mail server, supplied the event to READY.)

The span operators are very useful for constraining event sequences. For example, the following specification matches 5 MyAlarm events with Priority values that are all within 2 of each other.

```
{Ev1[5..5] MyAlarm | }
    where span_le(Ev1; $Priority; 2) ;
notify Ev1
```

Recall that Ev1 without a bracketed event selection stands for the sequence of 5 events. This sequence is used as the first argument of span_le and as the argument of the notify action.

6 Event Domain Boundaries

There must be a way to manage interactions between different event domains (different administrative domains, different application frameworks, and so on). In the normal case, the network infrastructure of each domain is disjoint. A boundary event router can sit at a junction between two or more named domains and manage the flow of events across the boundary. The types that are used, the conventions adopted for use of particular fields within an event type, *etc.*, can differ across domains, thus complex mapping capabilities are needed at this boundary.

Oki *et.al.* [14] observe that boundary routers can perform such mappings by having the router consume an event at one side of the boundary and supply another event on the other side. Mappings occur outside of the event service (and would be coded in a general purpose programming language). We chose to extend the power of our specification language to support boundary mapping specifications. In addition to providing a language for writing mappings, this approach provides support for dynamic control over the boundary router: as with normal specifications, it is easy to add, remove, suspend, or resume mapping specifications (and groups of mapping specifications).

Two extensions to the specification language are required. First, event types in mapping specifications must be augmented with domain names. Second, for an aggregation function or other local action that supplies a new event, a set of one or more domains for the supplied event must be specified. The common case is to specify one-to-one mappings, but compound matching and event aggregation can be used for more complex boundary cases.

An advanced feature of boundary routers is the ability to restrict network bandwidth used up by mapped events. With this feature, boundary routers turn out to be especially useful for carrying out a safe trial of a new service that relies on events generated by existing "released" services. A boundary mapping from trial event to released event can perform sanity checks to ensure that events will not bring down the released services. Further, bandwidth restrictions on mapped trial events shield the released services from serious performance degradation, *e.g.,* due to a bug in a trial service that generates too many events.

7 Conclusions

Event services are an important component of many application frameworks [6, 10]. Frameworks specify a consensus on common component types, event types and behavior patterns for the particular application domain (how do components invoke each other directly, when do components supply key events, *etc.*). Since this consensus will evolve over time, it is important to use an event service that supports event type evolution. By providing high-level constructs that describe expected patterns of behavior, more of a framework's specification can be captured and shared by applications. (Compound event specifications are a step in the right direction, but much more is needed.)

In this paper, we presented the high-level constructs provided by the READY event notification system. Compared to current event service products, READY offers a richer set of entity kinds (including grouping entities) and a more powerful set of operations over these entities. Not all event-driven applications can take advantage of this richer event model, but there are many application, from many application domains, that can. Our goal is to make this model available in a general-purpose event notification service.

A discussion of implementation issues requires another paper, but it is important to note here that providing high-level constructs does not imply that the service will be inefficient. In fact, the opposite is true: high-level declarations enable many optimizations that are not otherwise available. For example, high-level supplier descriptions, combined with high-level consumer specifications, enable low-level event-routing optimizations. In addition, common sub-expression elimination can be performed across all registered matching specifications, while common filters (capturing what is *not* interesting to any consumer) can be moved towards suppliers.

Regarding the implementation status, a version of READY that supports simple event matching and some of the high-level constructs mentioned in this paper is operational. The current version is being used as a core component of a unified messaging system in AT&T Labs. Currently, we are in the process of finishing the implementation of the compound matching component and incorporating into the system all of the constructs we have described.

Acknowledgments

Raghav Srinivasan provided useful comments on this paper, and is spending the summer helping with the READY implementation effort. The members of the Hermod unified messaging project participated in many interesting discussions motivating the need for READY.

References

[1] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using events to build distributed applications. In *Second International Work-*

shop on Services in Distributed and Networked Environments, pages 148–155. IEEE Computer Society Technical Committee on Distributed Processing, IEEE Computer Society Press, June 1995.

[2] A. Biliris, R. Gruber, G. Hjalmtysson, H.V. Jagadish, M.A. Jones, M.F. McGroary, E. Panagos, M. Rabinovich, A.W. Robinson, S. Spear, D. Srivastave, and D. Vista. HERMOD: A distributed database infrastructure for electronic messaging. Submitted for publication, 1998.

[3] Microsoft Corporation. Component object model (COM). http://www.microsoft.com/com/.

[4] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ladin, D. R. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC project: Combining active databases and timing constraints. *ACM-SIGMOD Record*, 17(1):51–70, March 1988.

[5] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *CACM*, 39(4), April 1996.

[6] M. Fayad and D. Schmidt. Object-oriented application frameworks. *CACM*, 40(10), October 1997.

[7] R. E. Gruber, B. Krishnamurthy, and E. Panagos. READY: A notification service for ATLAS. Technical report, AT&T Labs - Research, 180 Park Avenue, Florham Park, NJ 07932, September 1997.

[8] T.H. Harrison, D.L. Levine, and D.C. Schmidt. The design and performance of a real-time CORBA object event service. In *Proceeding of the OOPSLA '97 Conference, Atlanta, Georgia*, October 1997.

[9] R. Hayton. OASIS: An open architecture for secure interworking services. Technical report, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, March 1996.

[10] R. Johnson. Frameworks = (components + patterns). *CACM*, 40(10), October 1997.

[11] B. Krishnamurthy and D. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10), October 1995.

[12] C. Ma and J. Bacon. COBEA: A CORBA-based event architecture. In *USENIX COOTS'98*, Santa Fe, New Mexico, April 27–30 1998.

[13] Inc. NEON, New Era of Networks. NEONet 3.1. http://www.neonsoft.com/prods/index.html.

[14] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus — an architecture for extensible distributed systems. In *14th ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, December 1993.

[15] Object Management Group (OMG). Event Service specification. ftp://www.omg.org/pub/docs/formal/97-12-11.pdf.

[16] Object Management Group (OMG). Notification Service specification. ftp://ftp.omg.org/pub/docs/telecom/98-06-15.ps.

[17] R. Van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *CACM*, 39(4), April 1996.

[18] D. Skeen. The enterprise-capable publish-subscribe server. http://www.vitria.com/.

[19] Level8 Systems. EventWorks. http://www.level8.com/eventworks-wp.htm.

[20] Tibco. TIB/Rendezvous. http://www.rv.tibco.com/rvwhitepaper.html.

[21] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.