Specifying Timing Constraints and Composite Events: An Application in the Design of Electronic Brokerages

Aloysius K. Mok, *Member*, *IEEE*, Prabhudev Konana, Guangtian Liu, Chan-Gun Lee, and Honguk Woo

Abstract—Increasingly, business applications need to capture consumers' complex preferences interactively and monitor those preferences by translating them into Event-Condition-Action (ECA) rules and syntactically correct processing specification. An expressive event model to specify primitive and composite events that may involve timing constraints among events is critical to such applications. Relying on the work done in active databases and real-time systems, this research proposes a new composite event model based on Real-Time Logic (RTL). The proposed event model does not require fixed event consumption policies and allows the users to represent the exact correlation of event instances in defining composite events. It also supports a wide-range of domain-specific temporal events and constraints, such as future events, time-constrained events, and relative events. This event model is validated within an electronic brokerage architecture that unbundles the required functionalities into three separable components—business rule manager, ECA rule manager, and event monitor—with well-defined interfaces. A proof-of-concept prototype was implemented in the Java programming language to demonstrate the expressiveness of the event model and the feasibility of the architecture. The performance of the composite event monitor was evaluated by varying the number of rules, event arrival rates, and type of composite events.

Index Terms—Active databases, real-time databases, electronic brokerages, event specification, timing constraints.

1 Introduction

In many applications, such as stock trading, network management, and process control, systems need to react to events based on complex user preferences. These user preferences can be expressed as Event-Condition-Action (ECA) rules. Fundamental to such applications is the event specification and detection mechanism, which has been studied extensively in the context of active databases (ADB) [6], [5], [19], [22], [29], [35], [39], [40]. Several mechanisms have been proposed for event specification and detection in ADB; ODE [21] uses finite state automata, Snoop implemented in Sentinel [5] uses event graphs, and SAMOS [17] uses colored Petri-nets. One of the challenges in various mechanisms is the effective representation and detection of satisfaction and violation of timing constraints among complex events that has been studied in temporal logic [2] and real-time systems (RTS) [24], [30], [32]. This paper proposes a unique event specification and detection mechanism based on Real-Time Logic (RTL) [25], [32], a first-order temporal logic, that is primarily designed to represent various timing constraints in RTS within the ECA paradigm of ADB.

Our RTL-based event specification language provides distinct advantages and a significant step forward in active databases on many dimensions including event consumption mode flexibility, expressiveness (i.e., semantic clarity, ability to represent complex events and timing constraints, extensibility), specification of timing constraints and detection, resource bound prediction (i.e., extent of event history requirements), and ease of implementation and maintenance.

Expressiveness is an overarching criterion for evaluating our RTL-based language since it is designed to specify and detect timing constraints among events. Further, the event consumption mode of RTL, which is based on *event instances* rather than *event types*, provides flexible mechanism to specify correlations among events. Event languages such as Snoop [5] are based on event types which require explicit consumption policies such as *parameter context* [5] to deal with the semantics involving operators such as AND, OR, and SEQ.[1] As noted in [1], languages that require enumeration of all possible consumption policies are restrictive and hard to extend. Thus, our RTL-based event language without predefined consumption mode and the ability to specify timing constraints provide significant benefits such as extensibility and ease of implementation and maintenance.

- *A.K. Mok, C.-G. Lee, and H. Woo are with the Department of Computer Sciences, Taylor Hall 2.124, The University of Texas at Austin, Austin, TX 78712. E-mail: {mok, cglee, honguk}@cs.utexas.edu.*
- *P. Konana is with the Department of Management Science and Information Systems, CBA 5.202; B6500, Graduate School of Business, The University of Texas at Austin, Austin, TX, 78721. E-mail: pkonana@mail.utexas.edu.*
- *G. Liu is with SBC Technology Resources, Inc., 9505 Arboretum Blvd, Austin, TX 78712. E-mail: gliu@tri.sbc.com.*

Manuscript received 25 Sept. 2003; revised 2 Sept. 2004; accepted 22 Sept. 2004.
Recommended for acceptance by Luqi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0151-0903.

1. If there are two events E_1 and E_2, then disjunction, E_1 OR E_2, implies either E_1 or E_2 occurs, and conjunction, E_1 AND E_2, implies both E_1 and E_2 occur. Sequence operator, E_1 SEQ E_2, implies E_2 occurs after an occurrence of E_1.

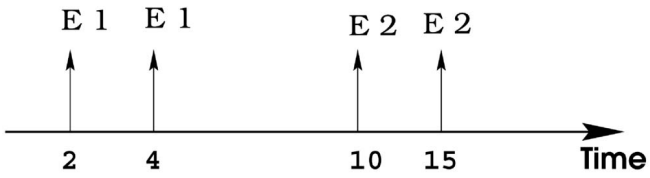


Fig. 1. An occurrence scenario of E_1 and E_2.

There are other direct benefits resulting from using RTL-based approach of RTS [25], [32], [34], [33]. First, our approach allows calculation of the maximum length of event history buffer (i.e., resource bound prediction) to be maintained for a given event specification (See Section 4.3). This property is a critical requirement for practical systems since it is unrealistic to allocate unbounded memory to event history buffer. Second, efficient event detection algorithms studied in RTL can be adopted where not only the satisfaction of timing constraint can be detected, but also the *violation* can be checked efficiently using derived implicit constraints from constraint graphs corresponding to the event specifications [34], [33].

Below, we demonstrate the contributions and relevance of our approach. Event operators such as AND, OR, and SEQ discussed in extant event description languages, including those in active databases, are used to define composite events by correlating events at the level of event types and not instances of events as proposed in this paper and in [1], [28], [29], [33], [34], [43]. The implications of such composite events are explained below with an example.

Consider the event instances of two events E_1 and E_2 shown in Fig. 1. Assume, a composite event $E = E_1\ AND\ E_2$ is expected to occur when both E_1 and E_2 have occurred. However, based on Fig. 1, at time 10 it is not clear, semantically, which event instances of E_1 must be associated with E_2 without additional information (i.e., semantics). When the first instance of E_2 occurs, it could be interpreted that two instances of E_1 to have occurred and trigger the same action twice. Even if only one action is triggered, there is an ambiguity as to which instance of E_1 should be considered.

In some special cases, it may be acceptable to let every possible event instance participate in forming the composite events. Let (e, i) denote ith instance of event e. At time 10, we can detect two instances of composite events, $\{(E_1,1),(E_2,1)\}$, $\{(E_1,2),(E_2,1)\}$ for $E_1\ AND\ E_2$. However, as indicated in [5], the system may suffer from the overhead of keeping huge event history and high cost of composite event detection. Therefore, this solution cannot be applied to general cases. Another solution is to limit eligible event instances in detecting composite events based on predefined event consumption policies [36] such as *parameter context* introduced in [5]. For example, when *recent* parameter context is used, at time 10 the most recent instance of composite event is detected; that is, $\{(E_1,2),(E_2,1)\}$ for $E_1\ AND\ E_2$. This requires that event specification must carry additional information about how to consume event instances. This approach may have advantages in situations where event consumption policies are enumerated before the deployment of a system and entails no further extensions. Section 3.4 provides additional examples to suggest the semantic richness available in our language since the explanation requires description of our event model first.

The other critical benefit of using RTL is its ability to not only exhibit logical correctness, but also the timing correctness that are typically defined as a series of timing constraints among events in RTS. RTL is a real-time temporal logic suitable for specifying various timing constraints including deadlines and delays on instances of events for time-critical systems. Since there can be recurrent event instances of the same event type, to distinguish each of them, RTL uses an occurrence index of the event instance. For example, a real-time system may require that there is a deadline timing constraint between a pair of event instances, the ith instance of event e_1 and the $i+3$th instance of event e_2.

Thus, in our approach, we combine the idea of ECA paradigm of active databases and the timing constraint based specification scheme from RTL to design a new composite event specification language.

We demonstrate the applicability of our event model through a proof-of-concept prototype within a broader architecture to *unbundle* active capability as a component that can be plugged into any system requiring active property [3], [20]. In order to demonstrate the applicability in practical settings, we propose an application-specific Trading Rule Language (TRL) to exhibit how the complexity of the event model can be hidden from users who interactively specify business rules. We believe the process of translation from TRL to the proposed event specification language demonstrates the ease of implementation and maintenance of the proposed mechanism. We implement a Java-based business rule manager, ECA rule manager, and event server. The event server implements the monitoring algorithms for composite event specifications as well as the compilation of timing constraints in the event specification for efficient detection. This Java-based prototype is tested for online trading application where customers can specify their preferences interactively for correctness and behavior.[2]

The paper is organized as follows: Related work is discussed in Section 2. In Section 3, the event specification model that unifies events in traditional active databases and timing constraints is discussed. The e-brokerage design architecture and the proof-of-concept prototype are discussed in Section 4. The results of proof-of-concept prototype evaluation of the e-brokerage for different event arrival rates and number of rules are discussed in Section 5. Finally, we conclude in Section 6 with directions for future research.

2 Related Work

As stated earlier, a number of ADB systems have been implemented: Ode [21], HiPAC [9], Areil [39], REACH [4], Starbust [42], Sentinel [5], and SAMOS [17]. A comparative

2. The purpose of the experimentation is not to compare efficiency or effectiveness with other mechanisms, but to demonstrate the correctness and the behavior.

study of some of these active databases is provided in [36]. All these ADB systems rely on predefined event consumption policies, which is different from our system. Recent parallel efforts described in [1], [43] address the problem of such predefined event consumption policies. A metamodel for specification of composite events is introduced in [43]. AMIT [1] presents a flexible way to define an event consumption policy using a combination of quantifiers and consumption conditions for each operand in a composite event. Thus, there is some emphasis to not rely on predefined event consumption policies in event specification languages.

Our event specification language relies on a subset of RTL. A survey of various temporal logic mechanisms for real-time systems specification is available in [2]. Temporal logic is a branch of modal logic for specifying time and order [11]. Traditionally, temporal logic has been primarily used to verify that a set of desirable properties comply with the system specification. Recent applications of temporal logic include temporal business rule checking and temporal security rule checking [12], [11]. Some commercial solutions based on temporal logic, e.g., TemporalRover and DBRover [10], are now available. They support Linear Time Logic (LTL) and take a similar approach to ours in that they do not incorporate a temporal verification functionality into database system monolithically, but implement it as a separate module so that it can be plugged into any database systems without major modifications. In [11], LTL augmented with time series (LTLD) assertions are used to monitor sequences of propositions over time series and it is shown that various properties like stability, ranges, and monotonicity can be asserted and monitored in runtime.

The applications discussed in recent work [12], [11], [10] using temporal logics LTL and MTL appears to be similar to the functionality of our prototype system for electronic brokerage (see Section 4). The main distinction is the support for advanced composite events in our approach. The main objective for those systems is to monitor a particular set of desired properties, which in turn can be represented as a set of conditions on data (e.g., stability, ranges, and monotonicity). Hence, in most cases, they do not need to keep event histories, unlike any systems supporting composite events. In contrast, the purpose of using RTL in our event specification language is to allow the users to describe appropriate actions upon specific situations where the corresponding composite events are detected and evaluated for conditions.

The Event-Condition-Action (ECA) paradigm originated from traditional Condition-Action (CA) style rules commonly found in production-rule systems. In fact, some active database systems including Areil [39] support CA rules without explicit specification of events. Production rules are processed in the context of recognition-act cycle [36]. In recognition cycle, rules containing satisfied conditions are identified and sent to the conflict set. A resolution process chooses one of the rules in the conflict set and fires it (the actions defined on the rule are executed). The cycle is repeated until there are no rules in the conflict set. RETE [13] and TREAT [31] are well-known algorithms to find candidate rules efficiently. Recently, Lee and Cheng [27] introduced a new technique which reduces the execution-time upper-bound of real-time rule-based systems by utilizing the dependencies among predicates. In [36], it is addressed that the techniques, such as RETE and TREAT, used for production-rule systems are not commonly used in ECA-based systems. Most of ECA-based systems support only ECA style rules; thus, the set of conditions need to be checked are restricted to the rules of which events are detected. In case an ECA-based system supports CA style rules as well as ECA rules, e.g., Ariel, then such techniques are still useful for improving the efficiency of matching process. Ariel [39] uses an extended and database-oriented RETE algorithm.

3 Event Model

In this paper, we assume events—things of interest—occur instantaneously at discrete points in time [6], [22], [35] rather than treating the event occurrence as an interval [38], [41], [16], [23], [14], [15]. An extensive description of event classification can be found in [6], [19], [22], [35]. In general, events are classified as primitive or composite events. Primitive events are predefined and cannot be broken down into any other events. Composite events are defined using logical operators, such as *conjunction* and *disjunction*. They can also have temporal dimension or timing constraints that can be modeled as part of the event specification. For example, we can have events that occur at specific time periods or have two events that occur within some time interval.

In our event model, composite events are defined with both *logical operators* and *timing constraints* (or temporal operators [35]) containing *event functions* on *event instances*. The event instances participating in triggering a composite event are explicitly determined by the event functions in the composite event definition. Notice that we distinguish *logical operators* from *event operators*. Logical operators, such as *conjunction* and *disjunction*, take Boolean type arguments and are used for connecting timing constraints in our event model. Event operators such as *AND*, *OR*, and *SEQ* take event type arguments.

Events can also be classified as internal or external events based on the origin of state changes. Internal events are database events caused by the insert, update, or delete operation of an object, or events that are related to transaction execution, such as begin, abort, or commit. External events are generated by the environment outside the database (e.g., ticker data from financial markets). Events can be *relative* to the occurrence of other events in which case they are referred to as *relative events*. To support time-based or temporal events, we can specify *future events* as suggested in [35].

Our composite event model is designed based on a subset of real-time logic (RTL). As discussed earlier, RTL is a first-order temporal logic and its primary purpose is to represent various timing constraints for real-time systems. RTL has the capability to specify absolute timing characteristics of real-time systems. The subset of RTL [26], on which our event specification language is built, has the following additional characteristics [7] over original RTL: 1) Every RTL formula consists of an integer and arithmetic inequalities between two terms. The integer represents a deadline

or a delay. Each term can be either a function or a variable. 2) Arithmetic expressions which have functions taking instances of themselves as arguments are not allowed. In general, it is known that the decision problem whether a safety assertion follows from a given specification in original RTL is undecidable. However, the subset of RTL allows us to use various graph algorithms for analysis. For example, we can construct the corresponding constraint graph from a set of formula in the subset of RTL and use efficient algorithms [34], [33] for monitoring the satisfaction and violation of the timing constraints.

3.1 Event Functions

An event e can be *recurrent*, that is, an event has multiple instances occurring at discrete time points t. An instance i of event e is the occurrence index in the history. An event e is associated with a set of attributes $Attr = \{A_1, A_2, \ldots, A_m\}$.

We can represent an instance of an event e as a tuple $(i, t, AttrVals)$ where i is an occurrence index, t is an occurrence time, and $AttrVals = \{V_1, V_2, \ldots, V_m\}$. V_k is a value of A_k in $Attr$ of e where $1 \leq k \leq m$. Therefore, the history of an event is a time-series of these instances represented as tuples. For simplicity, we use (e, i) to denote ith instance of event e.

We define event functions on event instances to represent the events and composite events (e.g., external events, relative events, temporal events, future events). We believe additional domain specific functions can be added and integrated into the event specification without significant effort.

Consider the example of online trading with three instances of ORCL stock event: $(1, 10, \{50\})$, $(2, 15, \{47\})$, $(3, 20, \{51\})$. The only attribute in ORCL event is *price*.

Definition 1 (@-function). *The occurrence time $@(e, i) \mid_t$ of event instance (e, i), where i is an integer representing the occurrence index, and t is the evaluation time.*

$$@(e,i) \mid_t = \begin{cases} \emptyset(Does\ not\ exist) & i \leq 0 \\ occurrence\ time\ of(e,i) & i > 0\ and \\ & (e,i)\ has\ occurred\ by\ time\ t \\ \infty(Not\ occurred\ yet) & i > 0\ and \\ & (e,i)\ has\ not\ occurred\ by\ time\ t. \end{cases}$$

For example, $@(ORCL, 1) \mid_{15}$ returns 10, the occurrence time of the event instance $(ORCL, 1)$. If $i \leq 0$, then (e, i) cannot be a real event instance; hence the function returns $\emptyset$ indicating that the corresponding event instance does not exist. On the other hand, ∞ is returned when the occurrence time of event instance (e, i) is after time t. Notice that the following order is assumed: $\forall t \in N, \emptyset < t < \infty$.

Definition 2 (#-function). *The most recent occurrence index (instance) i of event e at time t is defined as*

$$\#(e,t) = \begin{cases} 0 & 0 \leq t < @(e,1) \mid_t \\ i & @(e,i) \mid_t \leq t\ and\ @(e,i+1) \mid_t = \infty\ where\ i > 0. \end{cases}$$

For example, $\#(ORCL, 17)$ returns 2, the most recent occurrence index at time 17.

Definition 3 (Relative @-function). *The relative occurrence time, $@_r(e, T, i) \mid_t$, of event e is the occurrence time of the ith event instance from time T. This is equivalent to $@(e, \#(e, T) + i) \mid_t$.*

For example, $@_r(ORCL, 12, 1) \mid_{30}$ returns 15, the occurrence time of the event ORCL's next instance from time 12 based on event history at time 30.

Definition 4 (Attribute Function). *Given event e, an attribute $attr$ of event e, and an occurrence index i, we define $AttrValue(e, attr, i)$ = the value of attribute $attr$ of event instance (e, i)*

For example, $AttrValue(ORCL, Price, 2)$ returns 47, the value of the attribute "Price" of the event instance (ORCL,2).

Definition 5 (Relative Attribute Function). *Given event e, an attribute $attr$ of event e, a time point t, and an occurrence index i, we define*

$$AttrValue_r(e, attr, t, i) = AttrValue(e, attr, \#(e, t) + i).$$

For example, $AttrValue_r(ORCL, Price, 14, 2)$, returns 51, the value of the attribute "Price" of the second event instance from time 14.

Definition 6 (Periodic Timer Function). *Given a starting time t_1, periodic interval time p and an ending time t_2, we define periodic timer events $Ptime(t_1, p, t_2)$ that occurs at time $t_1 + j \times p$ for $j = 0, 1, .., n$ until time t_2.*

For example, a timer event is required every one hour starting at 8am and ending at 5pm, then the periodic timer event is $Ptime(8am, 1hr, 5pm)$ which initiates a timer event at 8am, 9am,...,5pm.

Definition 7 (Occurrence Function). *Given an event e and an instance i, we define occurrence Boolean predicate, $occ(e, i) \mid_t$, to signify whether the ith event instance has occurred or not by time t.*

3.2 Timing Constraints

In [9], [37], [35], it is noted that ECA rules can be used in active databases to define the timing constraints on various events. We can express timing constraints between the occurrence times of different event instances as a set of assertions. Timing constraints between event instances have applications in financial trading where consumers may relate an event from one market to another event in a different market (e.g., option market and security exchanges). Such timing constraints between two event instances can be represented as follows:

Definition 8 (General Constraint). *A general constraint specifies a timing relationship between two event instance occurrences. Generally, it can be expressed in the following form:*

$$T_1 + D\ op\ T_2,$$

where D is an integer constant, op is a relational operator, one of $\{>, \geq, <, \leq\}$, and T_1, T_2 are event instance occurrence

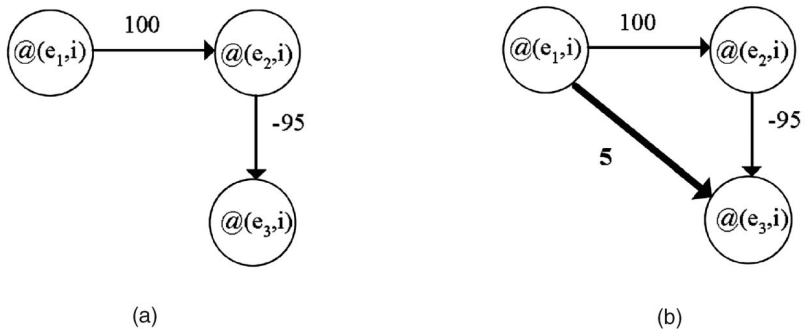


Fig. 2. (a) Original constraints. (b) Deriving an implicit constraint.

times, represented as the @-function, relative @-function, or 0*. By adjusting* D *and op, we can express any possible timing relationship between two events. The evaluation result of the above expression by applying the instances of events is a Boolean value.*

Example 1. With the general constraint[3]

$$@(e_1, i) + 10 \geq @(e_2, j),$$

we assert that the deadline for the jth instance of event e_2 to occur is 10 time units after the ith occurrence of event e_1.

Timing constraints in the form of general constraints can be imposed on various types of primitive and composite events including those related to transaction executions observed in real-time databases or real-time systems. For example, to assert that a transaction T should be completed (either aborted or committed) within 30 time units once it is started, we can use the following timing constraint expression:

$$\begin{aligned}&(occ(Start\_T, i) \wedge @(Start\_T, i) + 30\\ &\quad > @_r(Commit\_T, @(Start\_T, i), 1)) \vee\\ &(occ(Start\_T, i) \wedge @(Start\_T, i) + 30\\ &\quad > @_r(Abort\_T, @(Start\_T, i), 1)),\end{aligned}$$

where $Start\_T$, $Commit\_T$, and $Abort\_T$ are events denoting the start, commit, and abort of transaction T.

In practice, the following restricted class of general constraints is used in most cases:

Definition 9 (Simple Constraint). *A* ***simple constraint*** *is an expression of the form:* $T_1 + D \geq T_2$*, where* D *is an integer constant, and* T_1*,* T_2 *are time terms which may be an @-function, relative @-function, or* 0*, subject to the following restrictions:*

- *At least one of* T_1 *and* T_2 *must be a @-function.*
- *If* T_1 *is a relative @-function, its reference time must be* T_2 *and its occurrence parameter must be an integer constant, and vice versa.*

Generally, monitoring a set of timing constraints (TC) is not as simple as evaluating the truth-value of TC [8], [34], [33]. The violation of a TC is not simply verifying the satisfaction of the $\neg TC$, since at some point in time the truth value of the TC is unknown. This is particularly a challenging issue in the presence of conjunction of simple constraints that can lead to *implicit* constraints [29] as discussed in the example below:

Example 2. Assume we have timing constraints as follows:

$$@(e_1, i) + 100 \geq @(e_2, i) \; and \; @(e_2, i) - 95 \geq @(e_3, i).$$

The corresponding timing constraint graph is shown in Fig. 2a. In general, a timer is needed to detect timing violations as early as possible. For example, the constraint $@(e_1, i)+100 \geq @(e_2, i)$ will be violated 100 time units after the ith occurrence of event e_1 if the ith instance of event e_2 has not occurred by then. By starting a deadline timer expiring after 100 time units when (e_1, i) has occurred, we can catch the timing violation as soon as the violation is detectable.

Other than the need for the timer, implicit constraints which are hidden in the given timing constraints may contribute to detect violations as early as possible. Suppose the ith instance of event e_1 happens at time 0. If none of the ith instances of event e_2 and e_3 have occurred by time 5, then it is clear that the given timing constraints cannot be satisfied eventually. However, only with the given timing constraints, we cannot detect such violations until one of them is evaluated to show a violation. Now, suppose that we derive the implicit constraint $(e_1, i) + 5 \geq @(e_3, i)$ as illustrated in Fig. 2b. In this case, a deadline timer expiring after five time units is set as soon as (e_1, i) is occurred; hence, at time 5, we can detect a timing violation if the corresponding instance of event (e_3, i) does not occur by that time. If we do not exploit the implicit constraint, we would have to wait until time = 100 to catch a violation of the first timing constraint in case (e_2, i) does not occur by that time.

We present an expressive specification method for the composite event in Definition 10. Efficient TC monitoring algorithms required to catch the violations as early as possible are provided in [33], [34], which guarantee worst-case runtime complexity of $O(n)$ for detecting violation of simple timing constraints, where n is the number of timing constraints. Discussion about the detailed algorithm is beyond the scope of this paper.

3. For simplicity, we omit the evaluation time t when using event functions.

TABLE 1
Snoop Event Operators and Their Detection Conditions

Event Operator	Detection Condition
E_1 OR E_2	Occurs when either E_1 or E_2 occurs
E_1 AND E_2	Occurs when both E_1 and E_2 occur
E_1 SEQ E_2	Occurs when E_2 occurs after an occurrence of E_1
$P(E_1,t,E_3)$	Occurs periodically every t interval after an occurrence of E_1 until an occurrence of E_3
$P^*(E_1,t,E_3)$	Occurs when E_3 occurs and its occurrence time is at least t time after an occurrence of E_1
$A(E_1,E_2,E_3)$	Occurs when E_2 occurs within the time interval (E_1, E_3)
$A^*(E_1,E_2,E_3)$	Occurs when E_3 occurs after an occurrence of E_1

3.3 A Unified Specification Scheme

In this section, we present a unified scheme [29] for specifying composite events containing timing constraints. In this approach, a composite event is composed of the following constituents: attributes, timing constraints, TC, on other event instances, conditions, C, on attributes, and assignments to attributes. TC is in the format described in Section 3.2, and C is a Boolean expression involving $attribute\ functions$ and/or $relative\ attribute\ functions$. We refer to those events whose instances are referenced in TC and C as *component events*.

The representation and the triggering semantics of the composite event are provided in Definition 10.

An instance of the composite event CE having attributes

$$(Attrib_1 : type_1, Attrib_2 : type_2, \ldots, Attrib_n : type_n)$$

is triggered when TC is satisfied (or violated depending on its specification) and when C is true.

Definition 10 (Composite Event). *A composite event (CE) can be defined in the following format:*

DEFINE *CE* ***with attributes***
($[Attrib_1 : type_1], [Attrib_2 : type_2], \ldots, [Attrib_n : type_n]$*)*
OCCURRING WHENEVER
TC ***IS [SATISFIED | VIOLATED]***
IF *C*
THEN *assign values to CE's attributes*

CE's attribute assignments can be specified in the form CE.attrname := expression, where ***attrname*** *is the name of an attribute of CE, and* ***expression*** *is an arithmetic expression involving attribute functions and/or relative attribute functions of the component events.*

Example 3. We can define the following composite event ce, which will be triggered whenever IBM stock price drops more than 2 percent within 10 minutes and the same event was not triggered within the last 5 minutes:

```
DEFINE ce with attributes
   ([“b_price”:integer],[“Price”:double])
 OCCURRING WHENEVER
 @(IBM, i) - 5 minutes >=
    @_r(ce, @(IBM, i), 0) IS SATISFIED
IF AttrValue(IBM, “price”, i) <
    max(IBM, “price”, #(IBM, @(IBM, i) -
    10 minutes) + 1, i) * 0.98
THEN “b_price” := AttrValue(IBM,“price”,i)
```

where $max(e, attr, i_1, i_2)$ is a predefined function which gives the maximum value of e's attribute, **attr**, between the occurrence index i_1 and i_2.

In practical situations, the ability of referencing arbitrary event instances to the history may need special attention since we have limited resources (e.g., buffer). In Section 4.3, we present a technique to compute the maximum length of event history for a given specification. Thus, the system can utilize such information to reject an event specification and/or raise warning when the required event history exceeds the capacity of the system.

3.4 Using Event Functions on Event Instances

Table 1 provides the event operators defined in Snoop [6], a well-known composite event description language in ADB. An informal detection condition of each event operator is presented in this table. Notice that the event detection conditions for $A^*(E_1, E_2, E_3)$ and $E1\ SEQ\ E3$ appear to be similar based on the definitions in Snoop. While our event model does not provide predefined event operators, users can implement their own event operators having suitable semantics for their application purpose by defining composite events corresponding to the event operators. Before we present event expressions in our event model that corresponds to the event operators in Snoop, we briefly discuss the expression of the SEQ operator.

In [5], the operator defined as "$E_1\ SEQ\ E_2$ is detected when E_2 occurs provided E_1 has already occurred." In our system, if we adopt the detection condition of $E_1\ SEQ\ E_2$ as discussed in Snoop, then it would be $E_1\ SEQ\ E_2 \equiv @(E_1, j) < @(E_2, i) \wedge occ(E_2, i)$. However, as we have indicated, unrestricting the possible combinations of event instances in the event detection is not suitable for practical purposes because of unbounded event history. In our system, we can reasonably limit the

TABLE 2
Corresponding Composite Event Definitions

Event Operators	Corresponding Composite Event
E_1 OR E_2	$occ(E_1, i) \vee occ(E_2, j)$
E_1 AND E_2	$(occ(E_1, i) \wedge @_r(E_2, @(E_1, i), 0) > @_r(E, @(E_1, i), 0)) \vee$ $(occ(E_2, j) \wedge @_r(E_1, @(E_2, j), 0) > @_r(E, @(E_2, j), 0))$
E_1 SEQ E_2	$occ(E_2, i) \wedge occ_r(E_1, @(E_2, i), 0) \wedge @_r(E_1, @(E_2, i), 0) \geq @(E_2, i-1)$
$P(E_1,t,E_3)$	$\text{Ptime}(@(E_1, i)),\ t,\ @(E_3, i))$
$P^*(E_1,t,E_3)$	$occ(E_3, i) \wedge occ_r(E_1, @(E_3, i), 0) \wedge @_r(E_1, @(E_3, i), 0) \geq @(E_3, i-1)$ $\wedge @(E_3, i) \geq @r(E_1, @(E_3, i), 0) + t$
$A(E_1,E_2,E_3)$	$occ(E_2, i) \wedge occ_r(E_1, @(E_2, i), 0) \wedge occ_r(E_3, @(E_2, i), 1)$
$A^*(E_1,E_2,E_3)$	$occ(E_3, i) \wedge occ_r(E_1, @(E_3, i), 0) \wedge @_r(E_1, @(E_3, i), 0) \geq @(E_3, i-1)$

possible combinations by relating the event instances with a common event index variable and event functions. Depending on the applications, there can be different definitions for *sequence* operator. For example, we could define it as:

1. $E_1\ SEQ\ E_2 \equiv occ(E_2, i) \wedge occ_r(E_1, @(E_2, i), 0) \wedge @_r(E_1, @(E_2, i), 0) \geq @(E_2, i-1)$,
2. $E_1\ SEQ\ E_2 \equiv occ(E_1, i) \wedge occ(E_2, i) \wedge @(E_1, i) < @(E_2, i)$,
3. $E_1\ SEQ\ E_2 \equiv occ(E_2, i) \wedge @(E_1, 1) < @(E_2, i)$.

The semantics of 1, 2, and 3 are different in each case. The first case occurs when every instance of the event E_2 occurs provided an instance of the event E_1 has already occurred at or after the previous instance of the event E_2. The second case occurs when ith instance of the event E_2 occurs provided ith instance of the event E_1 has already occurred, where i is a positive integer. And, the last occurs when every instance of the event E_2 occurs provided any instance of the event E_1 has already occurred. Table 2 shows how event operators defined in *Snoop* can be represented in our event model. We assume that the *recent parameter context* is used for the above Snoop operators. While our event model does not have a concept of *context parameter*, users can specify the instance of an event they want explicitly by the event functions having the event instance argument. In addition to the event operators in the table, we define *within* event operator as follows:

$$E_1\ WITHIN\ t\ E_2 \equiv @(E_1, i) - t \leq @_r(E_2, @(E_1, i), 0) \vee @(E_2, i) - t \leq @_r(E_1, @(E_2, i), 0).$$

Since users can explicitly specify the occurrence index of the event instances in defining composite events under our event model, a user can easily express complex events like "$2i$th instance of E_1 and $7i$th instance of E_2 have occurred," where i is a positive integer. Although, business examples that require this type of composite event specification between event instances may not exist, it demonstrates the flexibility and richness of our event model. Additional examples of SEQ and A event operators are shown in Appendix B.

4 E-Brokerage Architecture

In many business contexts, such as online investing, auctions, and travel reservation, the value of information decays rapidly. Therefore, Internet-based electronic brokerages (e-brokerages) need to interactively capture consumer preferences as Event-Condition-Action (ECA) rules, and provide proactive services in a timely manner. In this ECA model, consumer desired actions (e.g., notification or market transactions) are executed when certain events occur in the business environment and some conditions become true.

In this section, we apply our composite event model to e-brokerages and present the design architecture. In addition, an implementation on the proposed architecture, which is based on client-server architecture, is illustrated. Fig. 3a illustrates an architecture of active real-time e-brokerage and Fig. 3b shows our implementation packages. Fig. 3c explains the internal details of the composite event monitor. The e-brokerage design architecture for providing active and real-time functionality consists of three subsystems: a business rule manager, an ECA rule manager, and an event server. The business rule manager provides intuitive rule specification constructs for consumers to specify their preferences interactively. It also provides consumers the rule management functions, such as add, delete, update, and conflict resolution. The business rule manager translates the preferences into ECA rules and passes them into the ECA rule manager where they are parsed and compiled. Since users may edit or delete their preferences frequently, it is advantageous to store rules in easily accessible form in the business rule base. This reduces the communication overhead and the time to reformulate ECA rules into user understandable form. During the compilation, the ECA rule manager specifies the composite events and stores the compiled rules into the rule base. The specified composite events are subscribed to the event server. The event server parses and compiles the composite event specifications and loads them into the composite event monitor (Fig. 3c) for monitoring purposes.

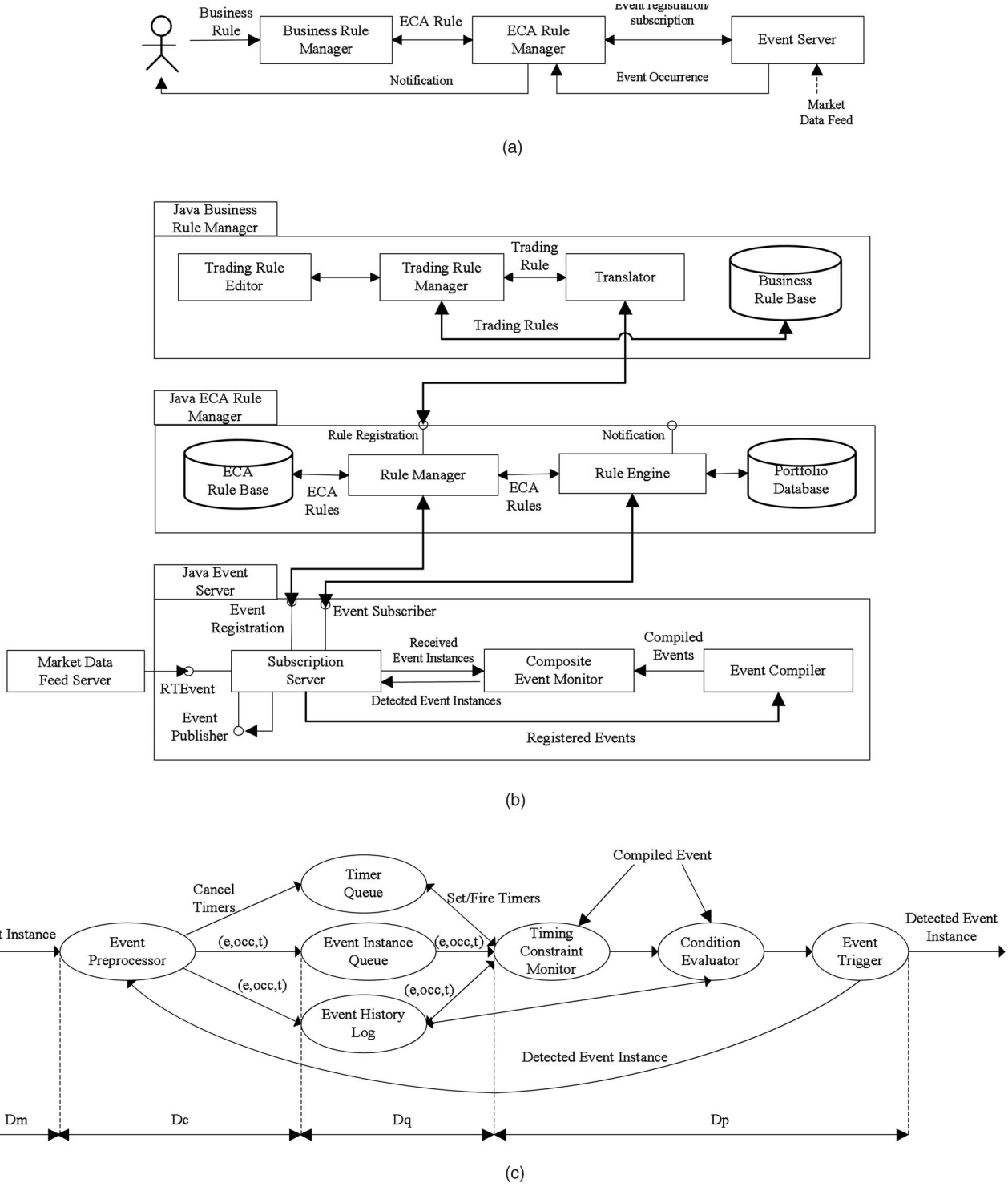


Fig. 3. (a) Active real-time e-brokerage architecture. (b) Active real-time e-brokerage package. (c) Composite event monitor.

The event server receives the external events as an event stream (e.g., stock market data feed). Whenever a subscribed event instance is received or detected by the event server, it is sent over to the ECA rule manager. Upon receiving such an event instance, the corresponding ECA rules (i.e., those which have this event specified in their event parts) are fired. If the condition parts—SQL queries against the database—are evaluated as true, then the action is taken (e.g., an e-mail alert or transaction execution).

The architecture provided has several advantages. The business rule manager is separated from the ECA rule manager and the event server, reflecting the true client-server nature of the architecture. The routine rule management functions are separated from the database, which makes e-brokerage maintainability and extendibility easier. The architecture separates the event monitoring from condition evaluation. There are two types of conditions: conditions on events and conditions on persistent data that

require access to the database (e.g., SQL queries). The evaluation of event conditions is performed within the event server while the evaluation of conditions on the persistent data is performed in the ECA rule manager. This provides greater flexibility when interfacing with different rule engines or databases. A simple publisher/subscriber interface provided in the event server allows communication between the event server and the ECA rule manager. Such separation allows us to adapt the composite event specification, compilation, and detection package to different applications and platforms easier.

4.1 Trading Rule Language (TRL)

In spite of the advantages of our approach presented earlier, we still need to consider many naive users who may not be familiar with such a formal event description language consisting of primitive yet powerful event functions. Another potential barrier is that our event specification language is primarily designed to express timing relationships between events; hence, it may not be an appropriate universal tool to capture user preferences. For example, a user's preference may include a condition which needs to query a database. For these reasons, we believe that it should be a reasonable approach to provide the end-users with an "easy" language which is tailored for specific applications.

As an example of such an "easy and application-specific" language, we introduce TRL (Trading Rule Language), which is primarily designed to facilitate the naive users to specify rules for stock market related activities. In TRL, user preferences are expressed in two conditions, one on market status and the other on portfolio status. Also, users can define actions which need to be taken upon the satisfaction of such conditions.

```
WHEN Conditions_On_Market_Status
IF Conditions_On_Portfolio_Condition
THEN Actions
```

Conditions_On_Market_Status specifies the conditions on market status including changes or current values of selected stock prices and market indexes. Two or more conditions are constructed by connecting pairs of conditions with predefined operators like SEQ, AND, OR, and WITHIN. Users can express specific conditions including market value changes of their holding stocks and gains/loses of their portfolio at *Conditions_On_Portfolio_Condition*. *Actions* represent a series of actions to be taken when such two conditions are met. Typical actions include a notification to a user and trading actions (buy or sell).

The following example shows how to use TRL to express trading rules for stock market.

```
WHEN (IBM.price increases by 1% within 1 hour)
  within 2 hour
  (previous 1 occurrence of
 (ORCL.price increases by 2% within 1 hour))
IF my_portfolio.gain > 20%
THEN { email "selling IBM";
    sell 50 IBM;}
```

We also provide a graphical user interface with which users can represent various rules in TRL. The event conditions expressed in *Conditions_On_Market_Status* are translated into our event specification language. There can be many ways to translate a *Conditions_On_Market_Status* into an event specification language due to various possible definitions of connectors. We discuss this issue in Section 4.3, and interested readers are invited to [28] for more details.

The previous TRL example is translated into the following event specification.

```
// Composite event for IBM increasing by 1%;
  Time is
// presented in milliseconds
  (1 hour = 3,600,000 milliseconds)
define ce1 = (["Occurrence Time" : time])
occurring whenever occ(IBM, i) is satisfied
if pred(IncreaseP, "IBM", "price", 0.01,
  3,600,000.0, "IBM")
then { "Occurrence Time" := @(IBM, i); };
// Composite event for ORCL declining by 2%
define ce2 = (["Occurrence Time" : time])
occurring whenever occ(ORCL, i) is satisfied
if pred(DecreaseP, "ORCL", "price", 0.02,
  3,600,000.0, "ORCL")
then { "Occurrence Time" := @(ORCL, i); };
// Composite event for the event part of
  the ECA rule
define ce3 = (["Occurrence Time" : time])
occurring whenever
    @(ce1, i) - 7,200,000 <= @r(ce2, @(ce1, i),
    0) is satisfied
if true then { ;};
```

Since *Conditions_On_Portfolio_Condition* contains application specific conditions (in this case user's portfolio condition), it is handled in an application server. It is stored as a condition against a result from an SQL query returning appropriate portfolio status (changes or value) in the application server. Only after the *Conditions_On_Market_Status* is evaluated as true by the event monitor, the application server checks *Conditions_On_Portfolio_Condition*. In case the condition is also evaluated as true, then the actions specified in the rule are taken. In the above example, the application server sends a notification to the user and places an order for selling 50 shares of IBM on behalf of the user.

4.2 Java Implementation and Composite Event Detection

The active and real-time component of the e-brokerage is implemented using Java. Oracle DBMS is used to store user portfolio data, business rules, and ECA rules. JDBC is used to communicate between Java programs and Oracle DBMS. From the perspective of software structure, the three subsystems in Fig. 3a correspond to three Java packages, JBRM (Java Business Rule Manager), JRM (Java ECA Rule Manager), and JEM (Java Event Manager), respectively. The dependency relationship among them and the internal details are shown in Fig. 3b.

Algorithm 1: Composite Event Detection Process
Input: $evtInst$: an event instance received
Output: none
CompositeEventDetectionProcess($evtInst$)
(1) Event Preprocessor Thread:
(2) Insert $evtInst$ into the concurrency control queue Q_1, based on $evtInst.occTime$;
(3) **if** $evtInst$ becomes header of Q_1
(4) **then** wake up the concurrency control thread;
(5) Concurrency Control Thread:
(6) **if** Q_1 is empty
(7) **then** $wait()$;
(8) **while** CurrentTime $\geq Q_1.header().occTime + ConcurrencyDelay$
(9) $wait(CurrentTime - Q_1.header().occTime - ConcurrencyDelay)$;
(10) $headerEvtInst \leftarrow Q_1.header()$;
(11) Add $headerEvtInst$ into its event history;
(12) Cancel all timers set for $headerEvtInst$;
(13) Append $headerEvtInst$ to the event process queue Q_2;
(14) Event Detector Thread:
(15) **if** Q_2 is empty **then** $wait()$;
(16) $theEvtInst \leftarrow Q_2.header()$;
(17) $evtList \leftarrow getRelatedCompositeEvents(theEvtInst)$;
(18) **foreach** composite event ce in $evtList$
(19) **if** $ce.checkTimingConstraint(theEvtInst) == true$
(20) **then if** $ce.checkCondition(theEvtInst) == true$
(21) **then** $ce.trigger(theEvtInst)$
(22)

Fig. 4. Algorithm 1.

The Java Event Manager implements the composite event specification, compilation, and detection discussed earlier. The JEM consists of three components: an event subscription server, an event compiler, and a composite event monitor. The subscription server implements the event registration, publisher, and subscriber interfaces. The subscription server receives primitive event registrations submitted by the event suppliers (e.g., market data feed) using the registration interface. Event consumers (e.g., ECA Rule Manager) use this interface to register composite events to the subscription server. When a composite event registration is received, the subscription server sends the composite event specification for compilation to the event compiler. The primitive and composite events are defined by using the event specification language discussed in Section 3. The compiled events are monitored in the composite event monitor. At runtime, event instances are received from the market data feed server through the RTEvent interface as shown in Fig. 3b. The subscription server receives these event instances and, if required, will pass those to the composite event monitor. All detected events are routed to the corresponding subscribers (rules) by the subscription server through the subscriber interface.

The details of the composite event monitor is given in Fig. 3c. An event instance is fed into the event monitor as a tuple $(i, t, Attr)$ with its name as discussed in Section 3.1. Each event instance is logged in the event history log as a tuple. Any outstanding timers set for the occurrence of this event instance is immediately canceled. This event instance is also put into the event instance queue for composite event detection. Since there may be delays from the occurrence of events to their arrival at the event preprocessor, this event should be processed by Concurrency Control Thread; that is, to make sure that at the time the header instance is processed, all event instances that occurred before it have been received and processed. The pseudocode for composite event monitoring is given in Algorithm 1 in Fig. 4.

Algorithm 2: Check the Satisfiability of a Constraint
Input: c: the constraint to be checked
Output: none
SCCHECK(c)
(1) **if** $c.T_2$ is known
(2) **if** $c.T_1$ is known
(3) **if** $c.T_1 + c.d \geq c.T_2$
(4) SATISFACTIONHANDLER(c)
(5) **else if** $c.T_2 - c.d \geq CurrentTime$
(6) Set delay timer for the event instance corresponding to $t_{ref}(c.T_1)$ and $c.T_1$ with timeout at $c.T_2 - c.d$
(7) **else**
(8) SATISFACTIONHANDLER(c)
(9) **return**

Fig. 5. Algorithm 2.

The representation of index numbers for long running systems can become an issue. A simple mechanism is to allocate sufficient bytes for representation to make the range large enough. However, in practice, the limited memory may hinder representation of large index numbers. In such cases, we can make index number circular; that is, the index can be wrapped. This solution is natural because, in general, the event history is implemented with a circular array in a fixed memory. To facilitate comparison of index numbers, the system remembers the last wrapped index number. In the case where all event instances are needed and wrapped index overwrites relevant ones, the system is compromised. Thus, it is crucial to assign the maximum index size carefully during the implementation of the system depending on application semantics. We can also consider a resetting signal to the system to restart the index number as long as the process does not hurt the integrity of the system.

4.3 The Maximum Length of Event History

It is required to maintain a history of each event, where the event instances are recorded for *checkTimingConstraint* function in Algorithm 1. Since the event detection thread in the algorithm is implemented to check satisfaction or violation of the timing constraints *only* when an event instance involved in the constraints newly arrives at the event process queue, it is possible to determine the bounded length of each event log.

For example, given a timing constraint $@(e_1, i) + 100 > @(e_1, i+1)$, it is possible to detect its satisfaction only with the latest two instances of e_1. In this case, simple arithmetic calculation on indexes of @-functions is needed to determine the necessary length of the event log of e_1. However, it is nontrivial to derive such property of bounded event history length, in general, for timing constraints involving instances of multiple events. Consider a timing constraint $c = @(e_1, f(i)) + d \geq @(e_2, g(i))$ and $r = maximum\ event\ occurrence\ rate\ of\ e_1\ and\ e_2$, where $f(i)$ and $g(i)$ are functions of event index i. The length of event logs $Len(event)$ is bounded during the satisfaction or violation detection of c:

- For satisfaction:
  $Len(e_1) = max(1, \lceil d \cdot r \rceil), Len(e_2) = 1.$
- For violation:
  $Len(e_1) = 1, Len(e_2) = max(1, \lceil -d \cdot r \rceil).$

Since we check c's satisfaction time (ST) $ST(c) = T_2$ according to the Algorithm 2 (Fig. 5), we only need to keep the current instance of e_2 in its log. If $d \geq 0$, at time $ST(c)$, we may need to access an instance of e_1 to know whether c is satisfied, so those instances of e_1 which occurred in the past d time units must be kept in its log. If $d < 0$, we need to set a delay timer which will expire when c is satisfied or cancelled by an instance of e_1 when c is violated. Hence, we need to maintain the outstanding timers within d time units rather than keeping all instances of e_1 in its log for checking satisfaction of c. Similarly, we can calculate the maximum length of event logs for the violation detection process.

However, for a timing constraint with a relative @-function to be checked appropriately, we need to be careful of event instances while the timing constraint reference time is being resolved. In [28], it was shown that the maximum length of event logs need not be necessarily bounded in the case of some *general constraints* with past relative @-functions.

Definition 11 (Reference Time). *Given a time term T, which could be an @-function, an $@_r$-function, or 0, we define its* ***reference time*** *$t_{ref}(T)$ as follows:*

$$t_{ref}(T) = \begin{cases} t & T = @_r(e, t, k) \\ 0 & T = 0 \vee T = @(e, i). \end{cases}$$

Definition 12 (OCP(T)). *Given a time term T which could be an @-function, $@_r$-function, or 0, we define function ocp(T) as follows:*

$$ocp(T) = \begin{cases} 0 & T \text{ is not a } @_r\text{-function} \\ |\text{occurrence parameter of } T| & T \text{ is a } @_r\text{-function}. \end{cases}$$

Suppose a *simple constraint* c: $T_1 + d > T_2$ and e_1 and e_2 are corresponding events for T_1 and T_2. If T_1 or T_2 is a past relative @-function, we check c for satisfaction at time $ST(c) = max(t_{ref}(T_2), T_2)$ because $t_{ref}(T_1) = T_2$ when T_1 is a relative @-function. Intuitively, when T_2 is a past relative @-function, in order to detect the satisfaction of c, we need to keep those instances of e_2 within its occurrence parameter while the reference time T_1 is being resolved. For the sake of brevity, the details are not provided here (see [28] for more details of the derivation). Finally, we can determine the maximum length of event logs at compilation time to detect the satisfaction or violation of a *simple constraint* c:

- For satisfaction:
  $Len(e_1) = max(1, \lceil d \cdot r \rceil), Len(e_2) = ocp(T_2) + 1.$
- For violation:
  $Len(e_1) = ocp(T_1) + 1, Len(e_2) = max(1, \lceil -d \cdot r \rceil).$

The existence of such a derivation method for calculating the maximum length of event logs implies that we can keep the length of event logs bounded, by analyzing constraints at the compilation time of composite events. At compilation time, the upper bound of length of the event log for an event e can be determined by the maximum length among all $Len(e)$ for timing constraints in which e is involved. In addition, any constant index of event functions regarding e is separately handled to be kept in its event log.

For an electronic brokerage architecture to be adapted in a wide-range of application domains, triggering conditions with complex timing correlations need to be incorporated with SQL-style functions mainly performing aggregate operations on a logical view of event instances in a certain range. In Example 3, the predefined function $MAX(e, attr, i_1, i_2)$ is used in the *if* clause of the ECA rule definition. Our focus is on composite event monitoring; hence, the condition and action parts in the rule language are intended to be simple in the implementation. Given such functions as $MAX(e, attr, i_1, i_2)$, however, calculation of the necessary length of event logs is largely a similar process to that of timing constraints as discussed so far. The study of designing functions of the condition part with efficient evaluations and yet high expressiveness is beyond the scope of this paper.

The bounded event log can be guaranteed by error checking mechanisms to restrict impractical time windows of users' rules. Our current prototype system, Java Business Rule Manager, includes the visual tool designed to help users to define their rules using a set of rule templates. While compiling a rule into a corresponding composite event specification, the length of all necessary event logs in the specification is computed and checked against the predefined values of system specifications.

5 Prototype Evaluation

We evaluated the functioning of the composite event monitor of our electronic brokerage under various scenarios to verify the correctness of event specification and detection and the behavior. Performance evaluation benchmark of active and real-time databases, in general, is still evolving [18] and difficult to compare across different systems because of a lack of objective measures related to expressiveness and extendibility. However, we rely on measures (e.g., detection latencies) discussed in [18] to fit our problem domain.

While manually checking correctness of event specification and detection of a large sample, we evaluated detection latencies under various conditions including event arrival rates, number of rules, and types of composite events formed by using different event operators (e.g., AND, OR, WITHIN, and SEQ). In the absence of a benchmark, the results may not provide information related to effectiveness or efficiency with respect to other methods; however, the behavior in terms of throughput of the composite event monitor on different event arrival rates and the saturation points of our system provide as assessment of the true functioning of the system. We believe our experimental setup provides a baseline for future studies in terms of generating events and composite events involving timing constraints.

5.1 Experimental Setup

The JEM, JRM, and JBRM were run on a PC with Pentium 733Mhz and 256MB memory, running Windows 2000. We used the socket utilities in the java.net to transmit event messages from the market data feed server to the monitor. All packages were written in JDK version 1.2.2.

Two issues are critical to the experimentation—the generation of valid rule sets, and the generation of event streams that affect the rule sets. Rules were generated with various types of events: simple rules and composite rules. Simple rules involve conditions on exactly one primitive event. We selected 74 companies (these companies were chosen by MBA students while they were participating in our field test) and defined primitive events with attributes: occurrence time, open price, high price, low price, current price, and volume.

Composite rules were generated based on how composite events are formed. We used four event operators—AND, OR, WITHIN, and SEQ—on two simple events to form a composite event. The rule generation program allows us to specify the number of rules to generate, and the proportion of rules related to a particular primitive event. The event firing rate is set to approximately 3 percent; for an incoming primitive event, about 3 percent of the rules are triggered. The program randomly generates rules by modifying the event name and adjusting the conditions of a predefined composite rule template. Fifty percent of the rules are simple rules, which means that they have only predicates on primitive events but containing no operators. The remaining 50 percent of the rules are composite rules, which have simple/composite events and event operators in them. The proportion of the event operators in the definitions are AND (5 percent), OR (10 percent), SEQ (10 percent), and WITHIN (25 percent).

The market data feed server was programmed to generate related events with randomly generated attribute

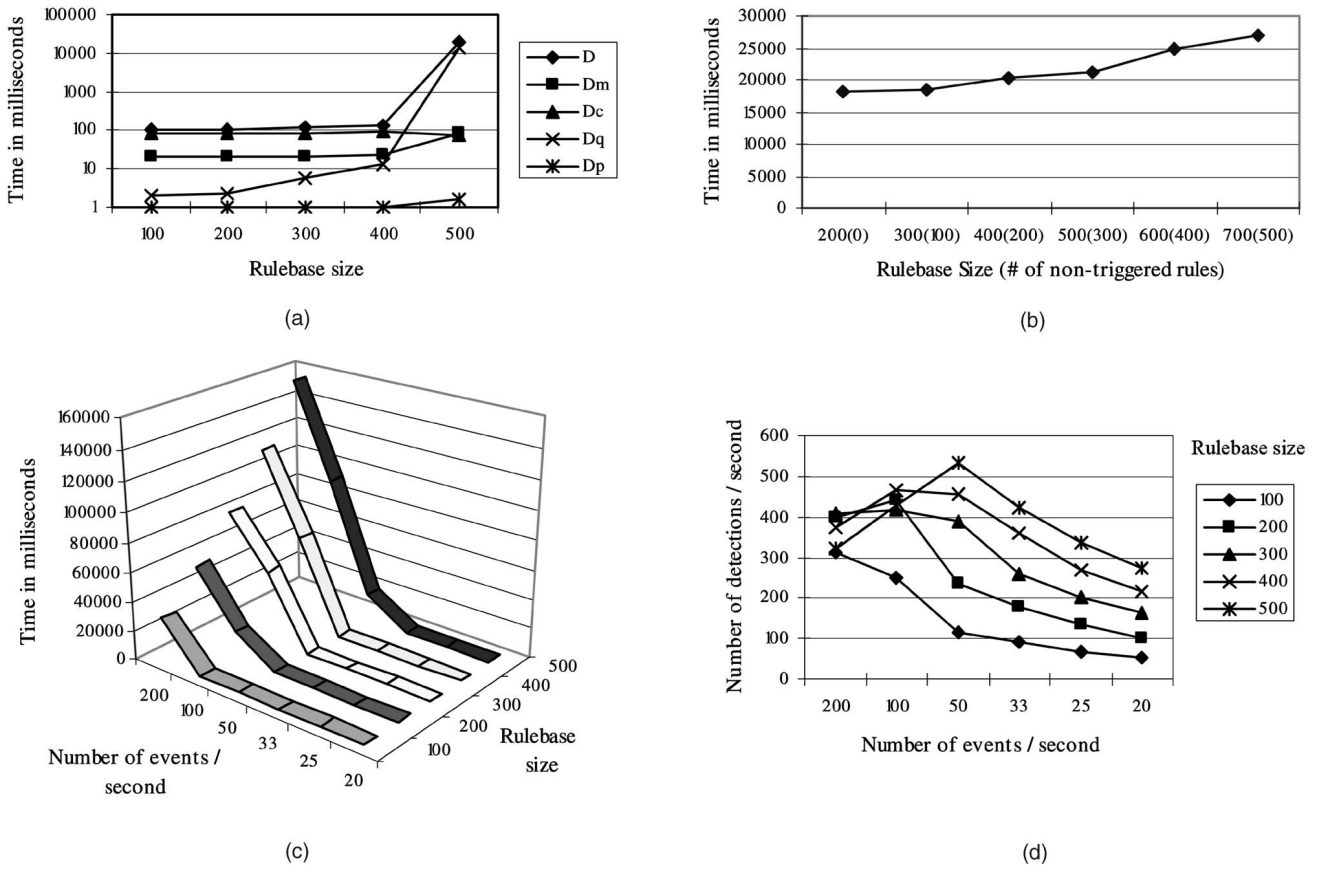


Fig. 6. Experiment Results: (a) Delay components. (b) Overhead of nontriggered rules. (c) Detection delay. (d) Throughput.

values from a predefined range that satisfy event conditions.

5.2 Experimental Results

Fig. 6a shows each component delay in the event detection. The X-axis represents the rulebase size (the number of rules) and the Y-axis represents the measured event detection delay in milliseconds. Note that the Y-axis is plotted in a log scale. Based on the composite event monitor shown in Fig. 3c, in this experiment we measure the following four delays:

- D_m—the component event message transmission delay between the market data feed server and the monitor server,
- D_c—the delay before a component event instance is placed in the queue for concurrency control,
- D_q—the queuing delay before a component event instance is ready to be processed, and
- D_p—the event processing time. The total detection latency D is the sum of the four delays.

The experiment was repeated to study the effect of increasing the number of rules under a constant event arrival rate. In each of the runs, the event arrival rate is fixed to 50 events/second. We increased the number of rules from 100 to 500. The detection latency D is fairly constant for up to 400 rules and then deteriorates rapidly. Concurrency delay D_c and event transmission delay D_m dominates until about 400 rules and then the queue delay D_q increases rapidly.

The second experiment reveals the overhead of non-triggered rules in the event monitor. In Fig. 6b, the X-axis represents the rulebase size. Each number in the parenthesis represents the number of nontriggered rules on the rulebase. The Y-axis represents the detection delay in milliseconds. While fixing the number of rules that will be detected to 200 and the event arrival rate to 100 events/sec, we increase the number of nontriggered rules to increase the workload on the event monitor. The graph shows that the overhead incurred by the nontriggered rules increases moderately with increase in the number of nontriggered rules. This is because nontriggered rules are filtered in the detection process and, therefore, do not participate in further event processing steps such as timing constraint checking, condition checking, and event triggering.

Fig. 6c shows the effect on the average detection latency under increasing number of rules and event arrival rates. In this figure, the X-axis represents the event arrival rate and the Y-axis represents the number of rules in the rule base. The Z-axis is detection latency in milliseconds. The results show that the detection latencies for all cases are fairly constant up to 33 events/sec. As the event arrival rate increases, the detection latency increases since the waiting queue at the event monitor increases. For the case of large

size rulebases, the saturation points appear sooner than those of small size rulebases.

Fig. 6d shows the throughput, the number of detected composite events per second. The X-axis represents the event arrival rate and the Y-axis represents the number of detected composite events per second. In this experiment, we are interested in observing the effect of overload in the system. Except for the case of the rule base size of 100, the overload points (where the number of detected composite events is decreasing while the event arrival rate is increasing) are clearly identifiable in each case. Once the system reaches overload point, as we have observed in the previous experiment, the detection latency and event queue continue to increase. Therefore, the number of detected composite events per unit time keeps decreasing. We expect the case of the rule base size of 100 will experience a similar effect in case we increase the event arrival rate sufficiently.

As discussed earlier, because of the lack of publicly available benchmark, we do not compare our event monitor performance with those of other event models. However, we can make several important observations. The event monitor specifies and detects composite events correctly in all the tests with a randomly generated set of composite events. The detection latency for the experiments conforms to the expected behavior. The performance can be improved simply by using faster processor. However, the critical issue is the scalability of the event monitor. In order to improve scalability, the queue delays must be reduced by balancing the event arrival rate and the processing time of events. If the event processing time cannot meet the demand of the event arrival rate, then the queue delay will increase dramatically. The event processing time can be improved by sharing events among different rules with which the event checking needs to be done only once. The issues to improve scalability and processing time are discussed under future research.

6 Future Research and Conclusion

While there is a rich literature of composite event specifications in ADB, we propose an alternative event specification model based on RTL. We present a unique approach to define composite events which does not require predefined event consumption policies. We accomplish this by combining the idea of ECA paradigm with timing-constraint based specification scheme of RTL. This feature provides extensibility of the model without having to make fundamental changes to the system implementation; that is, the extensibility provides certain level of easy maintenance of the system. Although it is difficult to objectively compare our model with extant event specification languages, through numerous examples (e.g., Appendix B and Section 3.4) we show the semantic richness of our model. In addition, we presented a method to calculate the maximum size of event history buffer to be maintained, an important characteristic of practical systems. Further, we relied on efficient event detection algorithms of RTL for the satisfaction and violation of timing constraints.

We demonstrated the practicality of our model with a prototype implementation of our event specification and detection for online trading application. The purpose of this prototype is not to compare the efficiency or the effectiveness with other event specification languages of ADB, but to evaluate the correctness and the system behavior under different input parameters. The system operates as expected and is found to correctly execute rules with different combinations of composite events with timing constraints.

In demonstrating our prototype, we proposed an architecture with unbundled active and real-time functionalities since monolithic systems are hard to extend and to maintain. The architecture consists of three separable components—business rule manager, ECA rule manager and event server—with well-defined interfaces.

There are a number of issues to be investigated further to make this prototype ready for commercial environment. We need to provide support for various coupling modes for transaction execution, which is not considered in this research. The next step is to develop algorithms for event sharing that enable scalability. The ECA rule manager should be able to share such primitive and composite events to gain additional efficiencies and to improve scalability. Apart from optimizing composite event detection algorithms and their implementation, efficient mechanisms for parallel processing to improve performance (detection latency) and scalability will be further investigated. The composite events can be partitioned into several *independent* subsets where each subset can be monitored separately, in parallel, on different processors. We are making arrangements to test the prototype with live data feed (e.g., stock market ticker data) in a true interactive environment. The issues of scalability and reliability can only be understood in such an environment.

Furthermore, the business rule manager will need to incorporate rule conflict resolution module of user rules since there can be legal implications of incorrect execution and noncompliance of rules. The approach being investigated is to push the burden of the final resolution of conflicting rules to the rule owner.

Appendix A

BNF Definition of Event Specification Language

The BNF definition is shown in Fig. 7.

Appendix B

Event Consumption Examples

B.1 Sequence (SEQ) Operator

Assume that we have two event types U and V. We denote e_i or (e, i) to express the $i\text{th}$ event instance of the event E. Suppose we have the following event history in increasing time order: $u_1, u_2, u_3, v_1, v_2, u_4, u_5, v_3, u_6, v_4$.

Let us consider the event definitions shown in Table 3. Each event definition can potentially represent a composite event U SEQ V (or $U; V$ in Snoop).

The following is the detection process of each event definition stated in Table 3.

```
⟨EventSpecs⟩   ::= { ⟨EventDef⟩ ; }
⟨EventDef⟩     ::= ⟨PEventDef⟩ | ⟨CEventDef⟩
⟨PEventDef⟩    ::= define primitive ⟨Name⟩ =
                   ( [ string : ⟨Type⟩ ] { , [ string : ⟨Type⟩ ] }* )
⟨Name⟩         ::= identifier
⟨Type⟩         ::= boolean | integer | long |
                   real | string | time
⟨CEventDef⟩    ::= define ⟨Name⟩ =
                   ( [ string : ⟨Type⟩ ] { , [ string : ⟨Type⟩ ] }* )
                   occurring whenever ⟨TC⟩ is ⟨Timing⟩
                   if ⟨Cond⟩ then { { string := ⟨Expr⟩ ; }* }
⟨Timing⟩       ::= satisfied | violated
⟨Truth⟩        ::= true | false
⟨TC⟩           ::= ⟨Conjunct⟩ { [ or ⟨Conjunct⟩ ] }
⟨Conjunct⟩     ::= ⟨TCTerm⟩ { [ and ⟨TCTerm⟩ ] }
⟨TCTerm⟩       ::= ⟨BConstraint⟩ | ⟨OccPred⟩ | ⟨Truth⟩
⟨BConstraint⟩  ::= ⟨TimeTerm⟩ ⟨AOP⟩ integer ⟨ROP⟩ ⟨TimeTerm⟩
⟨OccPred⟩      ::= occ ( ⟨Name⟩ , ⟨SimpOcc⟩ ) |
                   occ_r ( ⟨Name⟩ , ⟨ATFunc⟩ , integer )
⟨TimeTerm⟩     ::= ⟨AtFunc⟩ | ⟨RATFunc⟩ | 0
⟨ATFunc⟩       ::= @ ( ⟨Name⟩ , ⟨SimpOcc⟩ )
⟨RATFunc⟩      ::= @_r ( ⟨Name⟩ , ⟨ATFunc⟩ , integer )
⟨SimpOcc⟩      ::= integer identifier ⟨AOP⟩ integer |
                   identifier ⟨AOP⟩ integer |
                   identifier | integer
⟨Cond⟩         ::= ! ⟨Cond⟩ | ⟨Cond⟩ ⟨LOP⟩ ⟨Cond⟩ | ⟨BTerm⟩
⟨BTerm⟩        ::= ( ⟨Cond⟩ ) | ⟨Expr⟩ ⟨ROP⟩ ⟨Expr⟩ |
                   ⟨UserPred⟩ | ⟨Truth⟩
⟨Expr⟩         ::= ⟨Expr⟩ ⟨AOP⟩ ⟨Term⟩ | ⟨Term⟩
⟨Term⟩         ::= ⟨Term⟩ ⟨AAOP⟩ ⟨Term⟩ | ( ⟨Expr⟩ ) |
                   ⟨Funcs⟩ | ⟨Name⟩ | ⟨Number⟩
⟨Funcs⟩        ::= ⟨AtFunc⟩ | ⟨RATFunc⟩ | ⟨IndexFunc⟩ |
                   ⟨AttrFuncs⟩ | ⟨OtherFuncs⟩
⟨IndexFunc⟩    ::= # ( ⟨Name⟩ , ⟨TimeExpr⟩ )
⟨AttrFuncs⟩    ::= AttrValue ( ⟨Name⟩ , string , ⟨OccExpr⟩ ) |
                   AttrValue_r ( ⟨Name⟩ , string , ⟨TimeExpr⟩ , integer )
⟨OtherFuncs⟩   ::= max ( ⟨Name⟩ , ⟨OccExpr⟩ , ⟨OccExpr⟩ ) |
                   min ( ⟨Name⟩ , ⟨OccExpr⟩ , ⟨OccExpr⟩ )
⟨UserPred⟩     ::= pred ( ⟨Name⟩ , ⟨VParms⟩ )
⟨VParms⟩       ::= ⟨VParms⟩ , ⟨Expr⟩ | ⟨Expr⟩
⟨OccExpr⟩      ::= ⟨OccExpr⟩ ⟨AOP⟩ ⟨OccTerm⟩ | ⟨OccTerm⟩
⟨OccTerm⟩      ::= ( ⟨OccExpr⟩ ) | integer ⟨AAOP⟩ ⟨Name⟩ |
                   integer ⟨Name⟩ | integer | ⟨Name⟩ | ⟨IndexFunc⟩
⟨TimeExpr⟩     ::= ⟨TimeExpr⟩ ⟨AOP⟩ ⟨TimeExpr⟩ |
                   ⟨ATFunc⟩ | ⟨RATFunc⟩ | time
⟨AOP⟩          ::= + | -
⟨AAOP⟩         ::= * | /
⟨ROP⟩          ::= ≥ | ≤ | > | <
⟨LOP⟩          ::= and | or
```

Fig. 7. BNF definition.

TABLE 3
SEQ Operator

Event Definition	Detected Event Instances
(1) $occ_r(v, @(u,i), 1)$	$\{u_1, v_1\}, \{u_2, v_1\}, \{u_3, v_1\}, \{u_4, v_3\}, \{u_5, v_3\}, \{u_6, v_4\}$
(2) $@_r(u, @(v,i), 0) > @(v, i-1)$	$\{u_1, v_1\}, \{u_3, v_1\}, \{u_5, v_3\}, \{u_6, v_4\}$
(3) $occ_r(u, @(v,i), 0)$	$\{u_3, v_1\}, \{u_3, v_2\}, \{u_5, v_3\}, \{u_6, v_4\}$
(4) $@(u,i) \leq @(v,i)$	$\{u_1, v_1\}, \{u_2, v_2\}, \{u_3, v_3\}, \{u_4, v_4\}$
(5) $@(u,1) \leq @(v,i)$	$\{u_1, v_1\}, \{u_1, v_2\}, \{u_1, v_3\}, \{u_1, v_4\}$
(6) $@_r(v, @(u,i), 1) \geq @(u, i+1)$	$\{u_1, v_1\}, \{u_2, v_1\}, \{u_4, v_3\}$

1. Whenever u_i is detected, it tries to match the nearest future event V. As soon as such an event V is detected, then the composite event is fired.
2. Whenever v_i is detected, it tries to match the most recent event U. It is also needed that the occurrence time of the event U is no earlier than that of v_{i-1}, so that we do not use the event U repeatedly.
3. Whenever v_i is detected, it tries to match the most recent event U no matter whether the event U is used before.
4. Whenever v_i is detected, it tries to match the most recent event U of which index is same as that of the event V.
5. Whenever v_i is detected, it is matched to event u_1 if exists.
6. Whenever u_i is detected, it tries to find the matching future event V. It is also needed that the occurrence time of the event V is no later than that of u_{i+1}.

It is clear that item 1 is equivalent to $U;V$ in continuous parameter context (event consumption policy) in Snoop, item 2 is equivalent to $U;V$ in recent parameter context in Snoop. However, none of the event definitions items 3 to 6 is equivalent to $U;V$ with any parameter contexts in Snoop. The event representations items 1, 2, and 3 are possible and have nonspecial interpretations for the composite event U SEQ V. Each representation from items 4 to 6 has some additional requirements on top of the conventional SEQ operator. In item 4, we require an additional condition to match the occurrence indexes. In item 5, we relax the condition requiring the match of occurrence indexes, but at least u_1 should occur before, to match with any v_i. Item 6 excludes the cases where events U and V are immediately placed. At least one extra event U is needed between event U and event V.

B.2 Aperiodic (A) Event Operator

Now, suppose that we have the following event history in increasing time order:

$$u_1, u_2, u_3, v_1, v_2, w_1, w_2, v_3, u_4, w_3, w_4.$$

Let us consider the event definitions in Table 4. Each event definition can potentially represent a composite event A(U,V,W), which is triggered whenever V occurs within an interval [U,W].

Item 1 produces the instances same as the results of A(U,V,W) with continuous parameter context in Snoop. However, none of the A(U,V,W) with any parameter contexts in Snoop is equivalent to either items 2 or 3. We briefly explain why it is the case with the results from item 2. We can notice that u_3 is used for the output more than once. Clearly, it is not possible for this to happen in recent, chronicle, and cumulative parameter contexts of Snoop since it is required that we discard the event instance from the composite event buffer once it is used for a composition in such modes. If the continuous parameter context is applied, not only the triples with u_3 are included, but also the triples involving u_1 and u_2. This is because the continuous parameter context requires that every instance of U should be considered as a candidate for the beginning of the interval. A similar observation can be made for item 3.

ACKNOWLEDGMENTS

This paper integrates earlier conference proceedings papers from the Fourth IEEE Real-time Technology and Application Symposium in 1997, the 18th IEEE Real-Time Systems Symposium in 1997, the International Conference on Electronic Commerce and Telecommunications in 1998, and the 21st IEEE Real-time Systems Symposium in 2000.

TABLE 4
A Operator

Event Definition	Detected Event Instances
(1) $@_r(v, @(u,i), 1) < @_r(w, @(u,i), 1)$	$\{u_1, v_1, w_1\}, \{u_2, v_1, w_1\}, \{u_3, v_1, w_1\}$
(2) $occ_r(u, @(v,i), 0) \wedge occ_r(w, @(v,i), 1)$	$\{u_3, v_1, w_1\}, \{u_3, v_2, w_1\}, \{u_3, v_3, w_3\}$
(3) $@_r(u, @(w,i), 0) < @_r(v, @(w,i), 0)$	$\{u_3, v_2, w_1\}, \{u_3, v_2, w_2\}$

This work is supported by a US National Science Foundation CAREER Award under contract No. IIS-9875746 and an equipment grant from IBM.

REFERENCES

[1] A. Adi and O. Etzion, "Amit—the Situation Manager," *VLDB J.,* vol. 13, no. 2, pp. 177-203, 2004.

[2] P. Bellini, R. Mattolini, and P. Nesi, "Temporal Logics for Real-Time System Specification," *ACM Computing Survey,* vol. 32, no. 1, pp. 12-42, 2000.

[3] H. Branding and A.P. Buchmann, "Unbundling RTDBMS Functionality to Support WWW-Applications," *Proc. First Int'l Workshop Real-Time Databases,* pp. 45-47, Mar. 1996.

[4] A.P. Buchmann, A. Deutsch, J. Zimmermann, and M. Higa, "The REACH Active OODBMS," *Proc. 1995 ACM SIGMOD Int'l Conf. Management of Data,* p. 476, May 1995.

[5] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, "Composite Events for Active Databases: Semantics, Contexts and Detection," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 606-617, 1994.

[6] S. Chakravarthy and D. Mishra, "Snoop: An Expressive Event Specification Language for Active Databases," *Data and Knowledge Eng.,* vol. 14, no. 1, pp. 1-26, 1994.

[7] A.M.K. Cheng, *Real-Time Systems: Scheduling, Analysis, and Verification.* John Wiley & Sons, 2002.

[8] S.E. Chodrow, F. Jahanian, and M. Donner, "Runtime Monitoring of Real-Time Systems," *Proc. IEEE Real-Time Technology and Applications Symp. (RTAS),* pp. 74-83, 1991.

[9] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, and S. Sarin, "The HiPAC Project: Combining Active Databases and Timing Constraints," *ACM SIGMOD RECORD,* vol. 17, no. 1, pp. 51-70, Mar. 1988.

[10] DBRover, http://www.dbrover.com/, 2003.

[11] D. Drusinsky, "Monitoring Temporal Logic Specifications Combined with Time Series Constraints," *Proc. 15th Computer-Aided Verification Conf.,* pp. 114-117, 2003.

[12] D. Drusinsky, "The Temporal Rover and the ATG Rover," *Proc. Seventh Int'l SPIN Workshop SPIN Model Checking and Software Verification,* pp. 323-330, 2000.

[13] C.L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence,* vol. 19, no. 1, pp. 17-37, 1982.

[14] A. Galton and J.C. Augusto, "Two Approaches to Event Definition," *Proc. 13th Int'l Conf. Database and Expert Systems Applications (DEXA '02),* pp. 547-556, Sept. 2002.

[15] A. Galton, "Eventualities," *The Handbook of Time and Temporal Reasoning in Artificial Intelligence,* to be published.

[16] A. Galton, "Time and Change," *Handbook of Logic in Artificial Intelligence and Logic Programming,* vol. 4 (Epistemic and Temporal Reasoning), pp. 175-240, 1995.

[17] S. Gatziu and K.R. Dittrich, "SAMOS: An Active, Object-Oriented Database System," *IEEE Database Eng. Bull.,* vol. 15, no. 1, pp. 23-26, 1992.

[18] S. Gatziu, A. Gepperrt, and K.R. Dittrich, "A Designer's Benchmark for Active Database Management Systems: 007 Meets the BEAST," *Proc. Second Workshop Rules in Databases (RIDS),* pp. 309-326, Sept. 1994.

[19] S. Gatziu, A. Gepperrt, and K.R. Dittrich, "Detecting Composite Events in Active Dabase Systems Using Petri Nets," *Proc. Fourth Int'l Workshop Research Issues in Data Eng.,* pp. 2-9, Feb. 1994.

[20] S. Gatziu, A. Koschel, G. Bultzingsloewen, and H. Fritschi, "Unbundling Active Functionality," *SIGMOD RECORD,* vol. 27, no. 1, pp. 35-40, Mar. 1998.

[21] N.H. Gehani and H.V. Jagadish, "Ode as an Active Database: Constraints and Triggers," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 327-336, Sept. 1991.

[22] N.H. Gehani, H.V. Jagadish, and U. Shmueli, "Composite Event Specification in Active Databases: Model and Implementation," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 327-338, Aug. 1992.

[23] C.L. Hamblin, "Instants and Intervals," *The Study of Time,* pp. 324-328, 1972.

[24] F. Jahanian and A. Goyal, "A Formalism for Monitoring Real-Time Constraints at Runtime," *Proc. IEEE Fault-Tolerant Computing Symp.,* pp. 148-155, 1990.

[25] F. Jahanian and A.K. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Trans. Software Eng.,* vol. 12, no. 9, pp. 890-904, Sept. 1986.

[26] F. Jahanian and A.K. Mok, "A Graph-Theoretic Approach for Timing Analysis and Its Implementation," *IEEE Trans. Computers,* vol. 36, no. 8, pp. 961-975, Aug. 1987.

[27] Y.-H. Lee and A.M.K. Cheng, "Optimizing Real-Time Equational Rule-Based Systems," *IEEE Trans. Software Eng.,* vol. 30, no. 2, pp. 112-125, Feb. 2004.

[28] G. Liu, "An Event Service Architecture in Distributed Real-Time Systems," PhD thesis, The Univ. of Texas at Austin, 1999.

[29] G. Liu, A.K. Mok, and P. Konana, "A Unified Approach for Specifying Timing Constraints and Composite Events in Active Database Systems," *Proc. IEEE Real-Time Technology and Applications Symp. (RTAS),* pp. 199-208, June 1998.

[30] M. Mansouri-Samani and M. Sloman, "GEM: A Generalized Event Monitoring Language for Distributed Systems," *IEE/IOP/BCS Distributed Systems Eng. J.,* vol. 4, no. 2 June 1997.

[31] D.P. Miranker, "Treat: A Better Match Algorithm for AI Production Systems," *Proc. Sixth Nat'l Conf. Artificial Intelligence (AAAI-87),* pp. 42-47, Aug. 1987.

[32] A.K. Mok, "A Graph-Based Computation Model for Real-Time Systems," *Proc. IEEE Parallel Processing Conf.,* pp. 619-623, 1985.

[33] A.K. Mok and G. Liu, "Early Detection of Timing Constraint Violation at Runtime," *Proc. IEEE Real-Time Systems Symp. (RTSS),* pp. 176-185, 1997.

[34] A.K. Mok and G. Liu, "Efficient Runtime Monitoring of Timing Constraints," *Proc. IEEE Real-Time Technology and Applications Symp. (RTAS),* pp. 252-262, 1997.

[35] A.K. Patankar and A. Segev, "An Architecture and Construction of a Business Event Manager," *Temporal Database: Research and Practice,* pp. 257-280, 1998.

[36] N.W. Paton and O. Diaz, "Active Database Systems," *ACM Computing Surveys,* vol. 31, no. 1, pp. 63-103, 1999.

[37] K. Ramamritham, "Real-Time Databases," *Int'l J. Distributed and Parallel Databases,* vol. 1, no. 2, pp. 199-226, 1993.

[38] C.L. Roncancio, "Toward Duration-Based, Constrained and Dynamic Event Types," *Proc. Second Int'l Workshop Active, Real-Time and Temporal Database Systems (ARTDB '97),* vol. 1553, pp. 176-193, 1997.

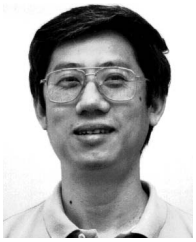
[39] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan, "Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* pp. 469-478, Sept. 1991.

[40] A.P. Sistla and O. Wolfson, "Temporal Conditions and Integrity Constraints in Active Database Systems," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 269-180, May 1995.

[41] P. Terenziani, "Is Point-Based Semantics Always Adequate for Temporal Databases," *Proc. Seventh Int'l Workshop Temporal Representation and Reasoning (TIME-00),* pp. 191-199, 2000.

[42] J. Widom and S. Ceri, "Active Database Rules," *Active Database Systems: Triggers and Rules For Advanced Database Processing,* 1996.

[43] D. Zimmer and R. Unland, "On the Semantics of Complex Events in Active Database Management Systems," *Proc. 15th Int'l Conf. Data Eng.,* pp. 392-399, 1999.



Aloysius K. Mok received the SB degree in electrical engineering, the SM degree in electrical engineering and computer science, and the PhD degree in computer science, all from the Massachusetts Institute of Technology. He is the Quincy Lee Centennial Professor in Computer Science at the University of Texas at Austin. Since 1983, he has been on the faculty of the Department of Computer Sciences at the University of Texas at Austin. He has done extensive research on computer software systems and is internationally known for his work in real-time systems. He is a past chairman of the Technical Committee on Real-Time Systems of the IEEE, and has served on numerous national and international research and advisory panels. His current interests include real-time and embedded systems, robust and secure network-centric computing, and real-time knowledge-based systems. He received in 2002 the IEEE Technical Committee on Real-Time Systems Award for his outstanding technical contributions and leadership achievements in real-time systems. He is a member of the IEEE.

Prabhudev Konana received the MBA and PhD degrees from the University of Arizona. He is an associate professor of MIS and a distinguished teaching professor at the McCombs School of Business, the University of Texas at Austin. His research interests are in electronic business value, electronic brokerages, online investor satisfaction, outsourcing and offshoring of IT and business processes, and online supply chain management. His research is supported by grants from a US National Science Foundation CAREER Award, US NSF Information Technology Research, Dell, Intel, and IBM. He has received numerous teaching awards. He has published more than 40 papers in journals and conference proceedings including *Management Science*, *Sloan Management Review*, *Management Information Systems Quarterly*, *Information Systems Research*, *Communications of the ACM*, *INFORMS Journal on Computing*, and *Information Systems*. He serves on the program committee of numerous conferences in the management of information technology and systems area.

Guangtian Liu received the BEng and MEng degrees in computer engineering from Tsinghua University, Beijing, China, and the MS and PhD degrees in computer science from the University of Texas at Austin in 1989, 1991, 1995, and 1999, respectively. He is currently a lead member of technical staff in SBC Laboratories, Austin, Texas. His current technical interests include network QoS, traffic engineering, performance monitoring, and network security.

Chan-Gun Lee received the BS degree in computer science from Chung-Ang University, Seoul, Korea, in 1996, and the MS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), Daejon, Korea, in 1998. Currently, he is a PhD candidate in the Department of Computer Sciences, University of Texas at Austin. His current research interests include real-time systems, streaming data monitoring, and real-time database systems.

Honguk Woo received the BS and MS degrees in computer science from Korea University and University of Texas at Austin in 1995 and 2002, respectively. He is currently a PhD student in the Department of Computer Sciences, University of Texas at Austin. Previously, he worked as a software engineer and consultant at KCC, IBM, and Samsung SDS for six years. His research interests include real-time event monitoring and real-time databases.