COMPOSITE EVENTS FOR DISTRIBUTED ENVIRONMENTS: SEMANTICS, ALGORITHMS AND IMPLEMENTATION

By

SHUANG YANG

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1999

Dedicated to my
Parents

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on this interesting topic and for providing me great advice, guidance and support through this research work.

I am extremely thankful to Dr. Joachim Hammer and Dr. Yuan Chieh Chow for agreeing to serve on my committee.

I would like to thank Sharon Grant for maintaining a well administered research environment with her indefatigable spirit and commitment to work and always being so helpful in times of need.

I am grateful to Hyoungjin Kim as well as all others in our group for invaluable help and fruitful discussions during the design and implementation of this work. I will also take this opportunity to thank all the graduate students at the database center for their help and friendship.

I would like to thank the Office of Naval Research and the Navy Command, Control and Ocean Surveillance Center RDT&E Division for supporting this work.

Last, but not the least, I thank my family for their love and support, especially my husband, Jingshun, and my brother, Lei. Also, I would like to thank all my true friends. Without their encouragement and endurance, this work would not have been possible.

TABLE OF CONTENTS

Abstract of Thesis
Presented to the Graduate School of the University of Florida
in Partial Fulfillment of the Requirements for the
Degree of Master of Science

COMPOSITE EVENTS FOR DISTRIBUTED ENVIRONMENTS:
SEMANTICS, ALGORITHMS AND IMPLEMENTATION

By

Shuang Yang

May 1999

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Sciences

Languages for event specification in centralized systems and their semantics have received considerable attention in the literature. In contrast, very little work exists on extending the semantics of event specification languages to distributed environments. This severely restricts the use of event-condition-action (or ECA) rules for a large number of distributed applications that are becoming prevalent such as change propagation in data warehouses. The difficulty of extending the event semantics to distributed cases are due to the special characteristics of the distributed environment, especially the lack of global time. In S. Schwiderski's dissertation, "Monitoring the Behaviour of Distributed Systems," a semantics of time stamps and their ordering based on the approximated global time and $2g_g-$restricted temporal ordering are defined. But the definition of composite time stamp ordering is logically inconsistent and suffers from lack of proof.

This thesis provides a formal framework of partial ordered sets for distributed composite event detection. A well-defined distributed composite time stamps and their *least restricted strict ordering* are defined under closed scrutiny and are carefully chosen based on mathematical reasoning to ensure the best semantics. The

concurrent and *weakened-less-then-or-equal* temporal relations are also introduced for the expressiveness of ECA rules, and furthermore, a *Max* operator is introduced for propagating the composite event time stamps. Based on this partial ordering and the Max operator on the time stamps, the semantics of Sentinel composite events is described for distributed event detection. The algorithms of the composite event operators are designed and implemented on the thesis.

CHAPTER 1
INTRODUCTION

Database management systems (DBMS) are designed to store, retrieve, update and analyze large volumes of data. Conventional database systems are *passive* which means the data is created, retrieved and updated in response to requests from users or the application programs outside the database system. The $demand-based$ characteristic of the passive DBMS cannot meet a large real-world situation which requires monitoring and reacting to the internal and external changes to the database state automatically without the intervention of the users or the application programs. *Active* DBMS enhances the functionality of the conventional DBMS by issuing the operations in response to certain event occurrences or conditions. This active capability is modeled by ECA (Event-Condition-Action) rules. When an event is detected and if the condition specified by the rules evaluates to be true, then an action is executed. Rule definition, event detection and action execution are some of the fundamental features provided by an active DBMS. Much work has been done on active functionality in the centralized context: Hipac [1], Ode [2, 3], ADAM [4],and SAMOS [5, 6, 7], Sentinel [8, 9, 10].

Many applications are distributed in nature and, hence, require active capability in a distributed environment. Most active DBMSs, so far, are dealing only with a uniprocessor or centralized environment. The difficulties in supporting ECA rules in distributed environments (as opposed to centralized systems) arise on account of: lack of global time, message delays between sites, and concurrent processes. The lack of the totally ordered global time makes it difficult to extend the ECA rules to distributed systems because the composite event detection relies heavily on the

total ordering of time. The message delays between different sites indicate the arrival order at event detectors and does not reflect the order of occurrence of the events in different sites. Different policies of dealing with delayed events will give rise to alternative ways of detecting composite events.

This thesis focuses on deriving a well-defined time stamp and its ordering relation in a distributed environment. We extend the semantics of Sentinel composite events to a distributed system. A distributed composite time stamp is defined to be the set of *maximum* primitive time stamps from the participating primitive ones. A well-defined partial ordering of the (set of) time stamps is defined carefully based on mathematical reasoning for providing the semantics. The $\sim$, $\stackrel{<}{\sim}$ relations are also introduced for dealing with some of the Sentinel operators.

The Max operator on the time stamps is also introduced for propagating the events. Finally, the full semantics of Sentinel distributed composite event detection is discussed. This thesis provides a fundamental framework of partial ordered sets relation for distributed event detection. A number of properties concerning the partial ordering is presented and formally proved. Some of the them are not so obvious due to the partial ordered property and set representation. The definition of the partial ordering of the time stamps can also be useful for distributed applications based on approximated global time system and $2g_g$-restricted order.

This thesis is organized as follows. Chapter 2 briefly reviews some of the related work, especially Schwiderski's dissertation [11]. The differences between that work and this paper are discussed. In Chapter 3, an overview of centralized composite event semantics in Sentinel is presented. Chapter 4 introduces the distributed time and temporal relation based on the global time and $2g_g-$ restricted ordering. The definition of composite time stamps and their ordering along with the Max operator is presented in Chapter 5 and the distributed composite event semantics of Sentinel is

derived. Chapter 6 introduces two kinds of evaluation policies and Chapter 7 contains conclusions.

CHAPTER 2
RELATED WORK

In additional to Sentinel, there are several efforts attempting to monitor the behavior of the distributed systems. Microsoft's COM (Component Object Model) [12] and CORBA (Common Object Request Broker Architecture)[13] provide some fundamental distributed event services but none of them have the notion of composite events. Schwiderski's dissertation [11] presents a general concept of primitive and composite event specification, event semantics and event detection in distributed systems which is based on the notion of approximated global time and $2g_g$-restricted temporal order.

In [11], the syntax of primitive and composite events is derived from the work of both active database systems and distributed debugging systems. The primitive events are site-related and include time events, data manipulation events, transaction events and abstract events. The composite events are made up of primitive and/or other composite events and event operators and are defined recursively. There are six event operators: conjunction, disjunction, sequence, concurrency, iteration and negation which can be applied to local and/or remote sites. Event parameters of a detected event are introduced to evaluate the condition and to execute the action of an ECA rule.

The semantics of primitive and composite events establishes when and where an event occurs and depends largely on the notion of physical time in distributed systems. When a primitive event occurs, a time stamp is allocated and associated with the primitive event which is represented as a tuple containing the information of the original site, approximated global time and local time. When a composite event

is detected, a set of time stamps are collected corresponding to the time stamps of the constituent primitive and composite events. The temporal ordering relations $<$ and $\sim$ are defined on the primitive and composite time stamps based on the $2g_g-$restricted temporal order. The structure and the handling of the time stamps in a distributed system are also discussed. A simplified version of the semantics of ordering and handling time stamps is introduced for some specific applications in order to build a easier and more efficient implementation.

On the event detection issue, the architecture and algorithms for the detection of composite events at system runtime are developed. The event detectors are distributed to arbitrary sites and composite events are evaluated concurrently. Two different policies of evaluation are considered: asynchronous and synchronous evaluation. Asynchronous evaluation is based on the ad hoc occurrences of the signaled event and is evaluated immediately on the arrival of the suitable event occurrences without blocking while the synchronous one will wait until all corresponding sites have been checked for relevant occurrences. The asynchronous evaluation is suitable for real-time application or the applications requiring fast response time while the synchronous one is suitable for applications requiring a high degree of consistency and reliability.

Our approach is similar to [11]. One difference is that we enforce the concurrency and "latest" properties in the definition of the time stamps. The philosophy behind our definition is that only the "latest" time stamps are considered and carried to form the set of the composite time stamp, which is corresponding to the concept of *t_occ* in centralized systems. Another major difference is the definition of temporal ordering $<$ on the (set of) time stamps of the composite events. Our definition of stricted $<$ ordering satisfies the *irreflective* and more importantly *transitive* properties which ensure a well-defined mathematical ordering unlike the one defined on [11]. Finally,

our time stamp operator, Max, is conceptually similar to their "joining" operators, but defined in a more precise way and is well-integrated with our definition of distributed composite time stamps, in which the "latest" and "concurrency" properties are ensured.

CHAPTER 3
THE CENTRALIZED COMPOSITE EVENT SEMANTICS IN SENTINEL

Sentinel is an active object-oriented DBMS which supports ECA rules mostly in a centralized environment. In this chapter, an overview of the centralized composite event semantics in Sentinel is presented.

3.1 Time in the Centralized System

Time in the centralized systems is totally ordered and can be represented as the (local) clock ticks of the (local) physical clock from some starting point. The following are the fundamental concepts about time which can be found in [14]:

Definition 3.1 (Physical Clock) : A physical clock *is a device for time measurement that contains a counter and a* physical oscillation *mechanism that periodically generates an event to increase the counter. The periodic event is called the* microtick *of the clock. The duration between two consecutive microticks is the* granularity *of the clock.*

Definition 3.2 (Local Time ticks) : A local time ticks *of the physical clock k represents a moment in time, counted as the number of microticks with granularity g_k since some starting time. This time ticks can be modeled mathematically as discrete non-negative integers. The local time ticks is a logical representation of the physical clock.*

Example 3.1 Let $g_k = 1/100s$. A microticks of 10^6 is $10^6 \times 1/100s = 10^4 s$ after the starting point of the system.

In the centralized system, time is totally ordered as the non-negative integers. This means for any given two time represented as the time ticks $t1$ and $t2$, the temporal relationship between these two can be: $t1 < t2$, $t1 = t2$, or $t1 > t2$.

3.2 Primitive Events in Centralized Active DBMS

An event is an instantaneous occurrence of interest which occurs at a specific point in time. Primitive events are those that are pre-defined in the system. There are three kinds of primitive events supported in Sentinel: database events, temporal events and explicit events.

- Database events correspond to database operations, such as data manipulation operations, transactions, or methods in object-oriented database. There are two events specified by the event modifiers [15], *begin-of* and *end-of*, can be associated with each database operation.

- Temporal events are the events related to the time. There are two kinds of temporal events: absolute and relative. An absolute temporal event is specified with an absolute value of time. A relative temporal event is specified with a reference time point and the offset. The reference point may be any event in Sentinel including the absolute temporal event.

- Explicit events are those that are detected along with their parameters by the application programs (outside the DBMS) and are only managed by the DBMS. The explicit events need to be registered with the DBMS system to be used as primitive events.

Each primitive event is associated with a time stamp, which indicates the time ticks of the occurrence of the event. That is $t\_occ(e)$ = clock ticks (with g_k) at the point when the event occurs, denoted clock(e).

Definition 3.3 (Time stamps in the Centralized system) *Let e be a primitive event, the time stamp of that event is the time occurrence of the event denoted by $T(e)$. Formally,*

$$T(e) = t\_occ(e) = clock(e)$$

Any two primitive events $e1$ and $e2$ with the corresponding time stamps $T(e1)$ and $T(e2)$ can be totally ordered based on the ordering of the time stamps.

Definition 3.4 (Temporal Orders of Centralized Time stamps) *Let $e1$ and $e2$ be any primitive events then the temporal order of these two events are defined as follows:*

1. *happen-before.* $e1$ *is said to be* happen-before $e2$ *if* $T(e1) < T(e2)$.
2. *simultaneously.* $e1$ *is said to be* simultaneously *with* $e2$ *if* $T(e1) = T(e2)$.
3. *happen-after.* $e1$ *is said to be* happen-after $e2$ *if* $T(e1) > T(e2)$.

Some of events are not allowed occur simultaneously, but there are some events that have to occur simultaneously with other events. The following are the assumptions of simultaneity of events:

1. Each non-temporal event has at least one temporal event happening simultaneously.
2. Each composite event (will be defined later) has at least one primitive event happening simultaneously.
3. No two database events can happen simultaneously.

3.3 Centralized Composite Event Semantics in Sentinel

Primitive events are the basic building blocks for developing an expressive and useful composite event specification language. Primitive events are single occurrences

of interest, while the composite events represent the complex pattern of a set of primitive events. Composite events are denoted as *event expression* formed by primitive events associated by the event operators. Conceptually, a *primitive event type* is the name of the interested primitive event while a *composite event type* is the name of a pattern of the set of interested events (including primitive and composite events) specified by the event expression. The *event type* is usually predefined or registered in the system. An *event occurrence* or *event instance* is the event being detected or monitored at run time. An event can be defined as follow:

Definition 3.5 (Centralized Event) An event E (either primitive or composite) is a function from the time domain onto the boolean values, True and False.

$$E : T \rightarrow \{True, False\}$$

given by

$$E(t) = \begin{cases} T(rue) & \text{if an event of type } E \text{ occurs at time point } t \\ F(alse) & \text{otherwise} \end{cases}$$

The negation of the boolean function E is denoted as $\sim$E which given a time point, denotes the non-occurrence of the event at *that* point.

In Sentinel, the composite event expression is defined recursively, by using a set of the primitive events, event operators and the composite events.

Let $E_1, E_2, \ldots, E_m$ be any events (primitive or composite), the semantics of the composite event operators in Sentinel is defined as follows:

1. **OR (∇):** Disjunction of two events E_1 and E_2, denoted $E_1 \nabla E_2$ occurs when E_1 occurs or E_2 occurs. Formally,

$$(E_1 \nabla E_2)(t) = E_1(t) \vee E_2(t)$$

2. **AND ($\triangle$):** Conjunction of two events E_1 and E_2, denoted $E_1 \triangle E_2$ occurs when both E_1 and E_2 occur, irrespective of their order of occurrence. Formally,

$$\begin{aligned}(E_1 \triangle E_2)(t) &= (\exists t_1)\ (E_1(t_1) \wedge E_2(t)) \vee ((E_1(t) \wedge E_2(t_1)) \\ &\quad \wedge (t_1 \leq t)\end{aligned}$$

3. **ANY:** The conjunction events, denoted by $Any(m, E_1, E_2, \ldots, E_n)$ where $m \leq n$, occurs when m events out of the n *distinct* events specified occur, ignoring the relative order of their occurrence. Formally,

$$\begin{aligned}ANY(m, E_1, E_2, \cdots, E_n)(t) &= (\exists t_1, t_2, \ldots, t_m) \\ &\quad (E_i(t_1) \wedge E_j(t_2) \wedge \cdots \wedge E_k(t_m)) \\ &\quad \wedge (t_1 \leq t_2 \leq \cdots \leq t_m \wedge t_m = t) \\ &\quad \wedge (i \neq j \neq \cdots \neq l)\end{aligned}$$

4. **Seq (;)** Sequence of two events E_1 and E_2, denoted $E_1;E_2$, occurs when E_2 occurs provided E_1 has already occurred. This implies that the time of occurrence of E_1 is guaranteed to be less than the time of occurrence of E_2. Formally,

$$\begin{aligned}(E_1; E_2)(t) &= (\exists t_1)(E_2(t) \wedge E_1(t_1)) \\ &\quad \wedge (t_1 < t)\end{aligned}$$

   It is possible that after the occurrence of E_1, E_2 does not occur at all. To avoid this situation, it is desirable that definite events, such as end-of-transaction or an absolute temporal event, are used appropriately.

5. **Aperiodic Operators (A, A*):** The Aperiodic operator A allows one to express the occurrence of an aperiodic event in the half-open interval formed by E_1 and E_2[1].

[1]The interval can either be (t_occ(E_1), t_occ(E_2)] or [t_occ(E_1), t_occ(E_2)).

There are two variants of this event specification. The *non-cumulative* variant of an aperiodic event is expressed as A(E_1, E_2, E_3), where E_1, E_2 and E_3 are arbitrary events. The event **A** is signaled each time E_2 occurs during the half-open interval defined by E_1 and E_3. **A** can occur zero or more times (zero times either when E_2 does not occur in the interval or when no interval exists for the definitions of E_1 and E_3). Formally,

$$A(E_1, E_2, E_3)(t) \quad = \quad (\exists t_1)(\forall t_2, t_1 \leq t_2 < t)(E_1(t_1) \wedge \sim E_3(t_2) \wedge E_2(t))$$

The *accumulative* version is denoted as $A^*(E_1, E_2, E_3)$ which occurs only once when E_3 occurs and accumulates all the occurrences of E_2 in the half-open interval formed by E_1 and E_3. This constructor is useful for integrity checking in databases and for collecting parameters of an event over an interval for computing aggregates. Formally,

$$\begin{aligned} A^*(E_1, E_2, E_3)(t) \quad &= \quad (\exists t_1)(E_1(t_1) \wedge E_3(t)) \\ &\quad\quad \wedge (t_1 < t) \end{aligned}$$

6. **Periodic Event Operators (P, P*):** A periodic event is a temporal event that occurs periodically. A periodic event is denoted as $P(E_1, TI[: parameters], E_3)$ where E_1 and E_3 are arbitrary events and $TI[: parameters]$ is a time interval specification with optional parameter list. P occurs for every TI in the half-open interval (E_1, E_3]. Formally,

$$\begin{aligned} P(E_1, TI[: parameters], E_3)(t) \quad &= \quad (\exists t_1)(\forall t_2)(E_1(t_1) \wedge ((t_1 \leq t_2 \leq t) \rightarrow (\sim E_3(t_2)) \\ &\quad\quad \wedge (t = t_1 + i * TI \text{ for some } 0 < i < t). \end{aligned}$$

P also has a accumulative version P* expressed as $P^*(E_1, TI[: parameters], E_3)$ which occurs only once when E_3 occurs. Also, specified parameters are collected

and *accumulated* at the end of each period and made available when P* occurs. Formally,

$$P^*(E_1, TI[: parameters], E_3) = (\exists t_1)(E_1(t_1) \wedge E_3(t)) \wedge (t \geq t_1 + TI)$$

Here TI is a time specification.

7. **Not (¬):** The not operator, denoted $\neg(E_2)[E_1, E_3]$ detects the non-occurrence of the event E_2 in the closed interval formed by E_1 and E_3. Note that this operator is different from that of !E (a unary operator in Ode [3]) which detects the occurrence of any event other than E.

$$\begin{aligned} \neg(E_2)[E_1, E_3](t) &= (\exists t_1)(\forall t_2)(E_1(t_1) \wedge \sim E_2(t_2) \wedge E_3(t)) \\ &\quad \wedge (t_1 \leq t_2 \leq t) \rightarrow \sim (E_2(t_2) \vee E_3(t_2))) \end{aligned}$$

The above operators is believed to meet the requirements of a large class of the applications including process control, networking. The operators have the following properties.

Proposition 3.1 1. **AND**, **OR** *are commutative, associative and distributive:*

$$A \triangle B = B \triangle B$$
$$(A \triangle B) \triangle C = A(\triangle B \triangle C)$$
$$A \nabla B = B \nabla B$$
$$(A \nabla B) \nabla C = A(\nabla B \nabla C)$$
$$A \triangle (B \nabla C) = (A \triangle B) \nabla (A \triangle C)$$

2. **AND** *can be derived from* **SEQ** *and* **OR**:

$$A \triangle B = (A; B) \nabla (B; A)$$

3. If $E3$ does not occur, **A** *and* **NOT** *are equivalent.*

$$A(E1, E2, E3) = NOT(E1, E3, E2)$$

CHAPTER 4
TIME IN DISTRIBUTED SYSTEMS

The composite event detection relies largely on on the comparison of the time of occurrence of events. Because of the lack of global time and lack of total ordering of the global time, a weakened semantics of approximated global time along with their partial ordering in distributed systems, and a semantics of distributed primitive time stamps and the partial ordering [11] are reviewed below. Besides, the notions of $\sim$, $\lesssim$ and "open", "closed" intervals are introduced for the expressiveness of Sentinel ECA rules. A number of properties about the time stamps and their ordering are discussed and proved and are used for the proofs of composite event ordering.

4.1 Ordering

Before introducing the global time and its temporal relation, it is necessary to define the *(strict) partial ordering* relation on any given set A in general set theory. This general definition is based on some of the important properties of a given relation on any set. Our definition of temporal ordering should follow this general definition of (partial) ordering relation to be a well-defined ordering relation. Also, this definition will serve as a basis when we define a well-defined ordering of given set.

Definition 4.1 *A relation R on a set A is said to be:*

reflexive *on A if* $(\forall x \in A)(xRx)$;

irreflexive *on A if* $(\forall x \in A)(\neg xRx)$;

transitive *on A if* $(\forall x, y, z \in A)(xRy \wedge yRz) \Rightarrow (xRz)$;

symmetric *on A if* $(\forall x, y \in A)(xRy \Rightarrow yRx)$;

asymmetric *on A if* $(\forall x, y \in A)(xRy \Rightarrow \neg(yRx))$;

antisymmetric *on A if* $(\forall x, y \in A)(xRy \wedge yRx) \Rightarrow (x = y)$;

The above properties are important when we define the ordering relation as well as when we evaluate an given ordering relationship. Based on the above properties, a general definition of *equivalent* , *(restricted) partial ordering* and *total ordering* can be defined as follow:

Definition 4.2 (equivalent relation) *A relation = on a set A is called a* equivalent relation *if it is transitive, reflexive and symmetric.*

Definition 4.3 (partial ordering and total ordering) *A relation $<$ on a set A is called a* strict partial ordering *if it is transitive and irreflexive.*
It is called a strict total ordering *if, in addition, we have:*

$$(\forall x, y \in A)(\ only\ x < y\ or\ x = y\ or\ y < x)$$

In centralized time system, the $<$ strict total ordering satisfies *irreflexive*, *transitive* and *asymmetric*, and the $\leq$ total ordering satisfies *reflexive*, *transitive*, and *antisymmetric*.

4.2 A Semantics of Global Time in Distributed Systems

The notion of physical time is a problem in distributed systems; there is no global time in nature. Each site in a distributed system has a single local physical clock with its own local clock tick which is transferred to local time by some software device. In order to compare the time of occurrence at the remote site, local clocks have to be synchronized. In a distributed system a global time can be achieved with synchronized local clocks through an *approximated global time base* [16, 17]. That is, there is a unique reference clock z with granularity g_z. The local clocks can be

synchronized by the concept of *precision* Π, which is the maximum time difference between two corresponding ticks of any two local clocks.

Definition 4.4 (Reference Clock) *Assume the existence of an omniscient external observer who can observe all events that are of interest in a given context. This observer is equipped with a unique reference clock z with frequency f^z which is in perfect agreement with the international standard of time. The counter of the reference clock is always the same as that of the international time standard. $1/f^z$ is called the granularity g_z of clock z.*

Because the granularity of the reference clock is so small, the digitalization error of the reference clock is usually be disregarded.

Given a primitive event e, $clock(e)$ denotes the local clock ticks perceived by the local system when the event occurs. $z(e)$ is called the *absolute time stamp* of the event e since z is the single reference clock in the distributed system.

Definition 4.5 (Precision) *Let $z(clock_k(i))$ be the time of occurrence of the $i-th$ tick in site k measured by the reference clock z. The* precision Π *is defined as follows:*

$$\Pi = Max\{\forall i \forall k \forall l : \|z(colck_k(i)) - z(colck_l(i))\|\}$$

The precision denotes the maximum offset of the time difference between the respective clock ticks in the interested clocks of the two clocks observed by the reference clock and measured as the ticks of the reference clock.

If all nodes in a distributed system could be perfectly synchronized with the reference time z which means $\Pi = 0$, then it would be easy to measure and compare any two primitive events by using the reference clock time as the time stamp for the events and the total ordering of the time stamps is easy to reconstruct. However, in a loosely coupled distributed system, every node has its own local clock ticks, such

a perfect synchronization of clocks in not possible. The concept of *global time* is introduced for a weakened notion of a universal time reference.

Given a set of nodes, each with its own local physical clock j and a granularity g_j. All the clocks are *internally synchronized with a precision* Π if for any two clocks j, k in the set of nodes and all microticks i in the reference clock, $|z(colck_k(i)) - z(colck_l(i))| < \Pi$.

Global time in a distributed system can be approximated by adjusting the granularity of local clock measurement to a global clock granularity g_g, that is, by selecting a subset of the microticks of each local clock j for generating the local implementation of a global notion of time. We need $g_g > \Pi$ to ensure that two simultaneous events receive time stamps distant at most $\leq 1g_g$. g_g can be chosen to be just greater than Π (for example, $g_g = \Pi + \varepsilon$) and then the global time of each site can be derived from the local clock ticks of the site.

Definition 4.6 (Global time) The global clock granularity g_g is given. The global time g_k of a local clock tick l_k is the local clock ticks expressed according to the standard (Gregorian) calendar with respect to some time zone (e.g. UTC, Universal Time Coordinated) and truncated to a global granularity g_g.

$$gt_k(lt_k) = TRUNC_{g_g}(clock_k(lt_k))$$

Here the "TRUNC" function could be *round* , *ceiling* or *floor* which depends on the application as long as it is consistent through out the system.

Example 4.1 Let j, k, l be a set of physical clocks in different sites of the Distributed system. Let z be the reference clock with granularity $g_z = 1/1000s$. Consider microtick of $i = 23991548127$. Assume:

$clockj$:

$granularity : g_j = 1/100s = 10g_z$

observed by the reference clock z : $z(clock_j(i)) = 239915481268$

$clockk$:

$granularity : g_k = 1/100s = 10g_z$

observed by the reference clock z : $z(clock_k(i)) = 239915481273$

$clockl$:

$granularity : g_l = 1/100s = 10g_z$

observed by the reference clock z : $z(clock_l(i)) = 239915481261$

Then $\Pi = 12g_z$.

Let $g_g = 100g_z = 10g_j = 10g_k = 10g_l$, *then the global time of microtick* i *is* 2399154812 *for all three physical clocks. Here TRUNC is integer division.*

Let $l_t(e)$ be the local ticks of the event e and $l_g(e)$ be the corresponding global time with global granularity g_g, then the ordering of the time in distributed system called $2g_g - precedence$ can be derived based on the local and global time of each site.

Definition 4.7 ($2g_g$-restricted temporal order) *The global clock granularity* g_g *is given.* $2g_g$*-restricted temporal order* $\rightarrow_{2g_g}$ *between primitive events* $e1$ *and* $e2$ *is defined as follows:*

1. *If* $e1$ *and* $e2$ *are primitive events occurring at the same site and* $l_t(e1) < l_t(e2)$ *then* $e1 \rightarrow_{2g_g} e2$.

2. *If* $e1$ *and* $e2$ *are primitive events occurring at the distinct sites and* $g_t(e1) < g_t(e2) - 1g_g$, *then* $e1 \rightarrow_{2g_g} e2$.

Definition 4.8 ($2g_g-$restricted concurrency) *The global clock granularity* g_g *is given.* $2g_g$*-restricted concurrency* $\|_{2g_g}$ *between primitive events* $e1$ *and* $e2$ *is defined as follows:*

$$e1\|_{2g_g}e2 \text{ iff } \neg(e1 \rightarrow_{2g_g} e2) \text{ and } \neg(e2 \rightarrow_{2g_g} e1)$$

Notice that the $2g_g$-restricted temporal order is irreflexive (ie, $e1 \rightarrow_{2g_g} e1$ is never true) and transitive (ie, if $e1 \rightarrow_{2g_g} e2$ and $e2 \rightarrow_{2g_g} e3$ then $e1 \rightarrow_{2g_g} e3$. Also, the $2g_g-$restricted concurrency relation is not an equivalent relation since it is not transitive. So, the $\rightarrow_{2g_g}$ is a valid strict partial ordering but not totally ordered which is different from the centralized system.

Example 4.2 *Let j, k, l be a set of physical clocks in different sites of the Distributed system. Let z be the reference clock with granularity $g_z = 1/1000s$. Suppose j, k, l have been synchronized with $g_g = 10g$, where $g = 1/100s$ is the granularity of the three clocks. Consider the following situation:*

$event : e1 :$ $clockj :$ $localtime : 23991548128$ $globaltime : 2399154812$

$event : e2 :$ $clockj :$ $localtime : 23991548129$ $globaltime : 2399154812$

$event : e3 :$ $clockk :$ $localtime : 23991548130$ $globaltime : 2399154813$

$event : e4 :$ $clockl :$ $localtime : 23991548140$ $globaltime : 2399154814$

Then :

$e1 \rightarrow_{2g_g} e2$, $e1 \|_{2g_g} e3$, $e1 \rightarrow_{2g_g} e4$.

Notice the "restrictiveness" of the temporal order. If $e1$ and $e2$ are in different sites, the they need at least $1g_g$ or > 10 local ticks in our example (depends on the TRUNC function) to guarantee the $\rightarrow 2g_g$ relation.

4.3 Distributed Time Stamps

Each event in the centralized system is associated with a time stamp (or even a counter which is advanced at the occurrence of each event) indicating when the event occurs. The detection of composite events is based on the ordering of the time stamp. In distributed systems, the time stamp of a global primitive event is a little more complicated [16, 17]. The information about the site, local time as well as the

the global time needs to be contained. Temporal relationship between time stamps can be derived directly from the definition of time stamps and the $2g_g - precedence$.

Definition 4.9 (Time stamps of the global primitive events) *A time stamp $T(e)$ of a global primitive event e with event type E is a function $T : E \rightarrow (site, global, local)$, where "site" is the site of occurrence of the primitive event, $local = l_t(e)$ and $global = g_t(e)$.*

If $T(e) = (site, global, local)$, then we also use the syntax of the object-oriented language to denote $site = T(e).site, global = T(e).global$ and $local = T(e).local$

Temporal relationship between time stamps can be derived from the time stamp and the $2g_g - precedence$.

Definition 4.10 (Temporal Relationship of Global Primitive Events) *On the basis of the $2g_g$-precedence time model, the temporal relationship between two time stamps $T(e1)$ and $T(e2)$ is defined as follows:*

1. *Happen-before:*

$$\begin{aligned} T(e1) < T(e2) \quad & iff \quad (T(e1).site = T(e2).site \wedge T(e1).local < T(e2).local)) \\ & \vee \quad (T(e1).site \neq T(e2).site \wedge T(e1).global < T(e2).global - 1g_g) \end{aligned}$$

2. *Simultaneous:*

$$T(e1) = T(e2) \ iff \ T(e1).site = T(e2).site \ and \ T(e1).local = T(e2).local$$

3. *concurrent:*

$$T(e1) \sim T(e2) \ iff \ \neg((T(e1) < T(e2)) \vee (T(e2) < T(e1)))$$

The following proposition demonstrate the relationship between the local time and the global time.

Proposition 4.1 *Let $T(e1)$ and $T(e2)$ be any two time stamps, then:*

1. *if $T(e1).local < T(e2).local$ then $T(e1).global \leq T(e2).global$*
2. *if $T(e1).local = T(e2).local$ then $T(e1).global = T(e2).global$*
3. *If $T(e1) \sim T(e2)$ and $T(e1).site \neq T(e2).site$ then*
   $T(e2).global - 1g_g \leq T(e1).global \leq T(e2).global + 1g_g$ and
   $T(e1).global - 1g_g \leq T(e2).global \leq T(e1).global + 1g_g$ or
   $|T(e1).global - T(e2).global| \leq 1$
4. *If $T(e1) \sim T(e2)$ and $T(e1).site = T(e2).site$ then $T(e1).local = T(e2).local$*

Proof.

1. If $T(e1).local < T(e2).local$ then $TRUNC_{g_g}(T(e1).local) \leq TRUNC_{g_g}(T(e2).local))$ by the property of the integer division.
2. If $T(e1).local = T(e2).local$ then $TRUNC_{g_g}(T(e1).local) = TRUNC_{g_g}(T(e2).local))$ by the property of the integer division.
3. $T(e1) \sim T(e2)$ and if $T(e1).site = T(e2).site$ then $T(e1).local = T(e2).local$ which means $T(e1).global = T(e2).global$, the result holds. If $T(e1).site \neq T(e2).site$ then $\neg((T(e1).global < T(e2).global - 1g_g) \vee (T(e2).global < T(e1).global - 1g_g))$, which implies $((T(e1).global \geq T(e2).global - 1g_g) \wedge (T(e2).global \geq T(e1).global - 1g_g))$. i.e., $((T(e2).global - T(e1).global \leq 1g_g) \wedge (T(e2).global - T(e1).global \leq 1g_g))$ which means $|T(e1).global = T(e2).global| \leq 1g_g$.
4. $T(e1) \sim T(e2)$ and $T(e1).site = T(e2).site$ then $\neg((T(e1).local < T(e2).local \vee T(e2).local < T(e1).local))$. So, $T(e1).local \geq T(e2).local \wedge T(e2).local \geq T(e1).local$ which means $T(e1).local = T(e2).local$.

□

The above indicates that the distinction between simultaneous and concurrent events is meaningful only for events from different sites.

The following proposition tells us the relation between the temporal order $\sim$ and $<$. Notice the differences between $=$ and $\sim$.

Proposition 4.2 *Let $T(e1)$, $T(e2)$ and $T(e3)$ be three time stamps, then*

1. *If $T(e1) = T(e2)$ and $T(e1) < T(e3)$ then $T(e2) < T(e3)$ regardless the sites of the events.*

2. *If $T(e1) \sim T(e2)$ and $T(e1) < T(e3)$ then $T(e2) < T(e3)$ does not hold.*

3. *If $T(e1) \sim T(e2)$ and $T(e2) \sim T(e3)$ then $T(e1) \sim T(e3)$ does not hold . ($T(e1).global = 1, T(e2).global = 2, T(e3).global = 3$ can serve as the counter example for the above two cases).*

Proof. $T(e1) = T(e2)$ implies $T(e1).site = T(e2).site$ and $T(e1).local = T(e2).local$ So, we have $T(e1).local = T(e2).local < T(e3).local$ if T(e3) site is same and $T(e1).global = T(e2).global < T(e3).global - 1g_g$ if T(e3) site is different. The result holds. □

Now we are ready to prove the strict partial ordering of $<$:

Theorem 4.1 (strict partial ordering of $<$) *The $<$ relation defined above is irreflexive and transitive, so it is a strict partial ordering relation on the set of distributed primitive time stamps.*

Proof. First, prove the irreflectivity, i.e., $\neg(T(e) < T(e))\forall T(e)$.
It easy to see it is true since $\neg(T(e).local < T(e).local)$ by the irreflectivity of the total ordering of local time.

Second, prove the transitivity, i.e., if $T(e1) < T(e2), T(e2) < T(e3)$ then $T(e1) < T(e3)$.

$$T(e1) < T(e2) \Rightarrow \{ \begin{array}{ll} T(e1).local < T(e2).local & \text{if } T(e1).site = T(e2).site. \\ T(e1).global < T(e2).global - 1g_g & \text{if } T(e1).site \neq T(e2).site. \end{array}$$

$$T(e2) < T(e3) \Rightarrow \{ \begin{array}{ll} T(e2).local < T(e3).local & \text{if } T(e2).site = T(e3).site. \\ T(e2).global < T(e3).global - 1g_g & \text{if } T(e2).site \neq T(e3).site. \end{array}$$

Case 1: $T(e1).site = T(e2).site = T(e3).site$. The conclusion holds, trivially.

Case 2: $T(e1).site \neq T(e2).site \neq T(e3).site$ (including the two cases $T(e1).site = T(e3).site$ and $T(e1).site \neq T(e3).site$). It is also very easy to prove.

Case 3: $T(e1).site = T(e2).site \neq T(e3).site$. Then we have:

$$T(e1).local < T(e2).local \Rightarrow T(e1).global \leq T(e2).global$$

and $T(e2).global < T(e3).global - 1g_g$. So we can conclude: $T(e1).global < T(e3).global - 1g_g$, which implies $T(e1) < T(e3)$.

Case 4: $T(e1).site \neq T(e2).site = T(e3).site$.

Then we have: $T(e1).global < T(e2).global - 1g_g$ and

$$T(e2).local < T(e3).local \Rightarrow T(e2).global \leq T(e3).global$$

So we have $T(e1).global < T(e2).global - 1g_g \leq T(e3).global - 1g_g$ which means $T(e1).global < T(e3).global - 1g_g$. The conclusion holds. □

The above theorem ensures a well-defined ordering definition. But that $\sim$ relationship is NOT transitive, so it is NOT a equivalent relation while the simultaneous relation $=$ is an equivalent relation. The definition of Simultaneous is a special case of Concurrent when sites are same.

Based on the definition of temporal order of the time stamps in distributed time system, a notion of *plus* of the time stamp, *open* and *closed* interval formed by the time stamps can be defined as follows:

Definition 4.11 (Plus) *Let T(e) be any time stamps of global primitive event e with local granularity g_k . Then $T(e) + T$ is defined as follows:*

$$T(e) + T = (T(e).site, T(e).global + Tg_g, T(e).local + T_k)$$

where T is the number of the units of global time with granularity g_g to be added and T_k is the calculated local time with g_k which is equivalent to T.

For example:

Example 4.3 *Let $T(e)$ be a primitive time stamp, with $T(e) = \{site1, 6, 65\}$. Suppose $g_g = 1/10sec$ and $g_k = 1/100sec$. Then $T(e) + 5sec = \{site1, 6 + 5 * 10, 65 + 5 * 100\}$.*

Definition 4.12 (weakened Less-than-equal relation for Primitive Time stamps) *Let T(e1), T(e2) be the time stamps of global primitive events $e1$ and $e2$. T(e1) is said to be* less-than-equal *to T(e2) denoted $T(e1) \lesssim T(e2)$:*

$$T(e1) \lesssim T(e2) \text{ iff } (T(e1) < T(e2)) \vee (T(e1) \sim T(e2)).$$

Definition 4.13 (Open Interval of Primitive Time stamps in Distributed Systems) *Let T(e1), T(e2) be the time stamps of global primitive event $e1$ and $e2$. An event e with time stamp T(e) is said to be in the* open interval *formed by $T(e1)$ and $T(e2)$ with $T(e1) < T(e2)$ denoted $T(e) \in \tilde{(}\ T(e1), T(e2)\ \tilde{)}$ is defined as follow:*

$$T(e) \in \tilde{(}\ T(e1), T(e2)\ \tilde{)} \text{ iff } T(e1) < T(e) < T(e2).$$

Suppose $T(e1), T(e2)$ are in different sites, then $T(e1) < T(e) < T(e2)$ implies:

$$\begin{aligned}
& T(e1).global && < T(e).global - 1g_g \text{ and} \\
& T(e).global && < T(e2).global - 1g_g \\
\Longrightarrow\ & T(e1).global && < T(e).global - 1g_g < T(e2).global - 2g_g \\
\Longrightarrow\ & T(e1).global && < T(e2).global - 3g_g \text{ for a non-empty open interval}
\end{aligned}$$

Intuitively, the open interval

$$\tilde{(}\ T(e1).global, T(e2).global\ \tilde{)}\ \text{ includes the points:}$$

$$\{T(e1).global + 2g_g, T(e1).global + 3g_g, \ldots, T(e2).global - 3g_g, T(e2).global - 2g_g\}.$$

Definition 4.14 (Closed Interval of Primitive Time stamps in Distributed Systems) *Let T(e1), T(e2) be the time stamps of global primitive event* $e1$ *and* $e2$. *A time stamp T(e) is said to be in the* closed interval *formed by* $T(e1)$ *and* $T(e2)$ *(requires* $T(e1) \lesssim T(e2)$ *) denoted* $T(e) \in \tilde{[}\ T(e1), T(e2)\ \tilde{]}$ *is defined as follow:*

$$T(e) \in \tilde{[}\ T(e1), T(e2)\ \tilde{]} \quad \textit{iff}\ T(e1) \lesssim T(e) \lesssim T(e2).$$

Suppose $T(e1), T(e2)$ are in different sites, then $T(e1) \lesssim T(e) \lesssim T(e2)$ implies:

$$|T(e1).global - T(e).global| \leq 1g_g \text{ and}$$

$$|T(e).global - T(e1).global| \leq 1g_g \text{ and}$$

$$\Longrightarrow \qquad |T(e1).global - T(e2).global| \leq 1g_g \text{ or}$$

$$T(e1) \sim T(e2) \text{ for a non-empty closed interval}$$

Intuitively, the closed interval

$$\tilde{[}\ T(e1).global, T(e2).global\ \tilde{]}\ \text{ includes the points:}$$

$$\{T(e1).global - 1g_g, T(e1).global, \ldots, T(e2).global, T(e2).global + 1g_g\}.$$

The *open* and *closed* intervals of given two time stamps of events is showed in the Figure 4.1.

There are some interesting properties about the $\lesssim$ semantics which are very useful when we derive the semantics of the time stamps of distributed composite events:

Proposition 4.3 *Let* $T(e1)$, $T(e2)$ *and* $T(e3)$ *be any three primitive time stamps, then:*

1. *(asymmetric) If* $T(e1) < T(e2)$, *then* $\neg(T(e2) < T(e1))$.

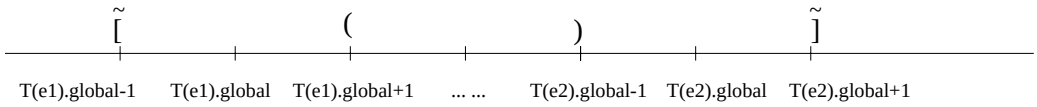

Figure 4.1. Open and Closed interval formed by T(e1) and T(e2)

2. *(antisymmetric) If* $T(e1) \lesssim T(e2)$ *and* $T(e2) \lesssim T(e1)$ *then* $T(e1) \sim T(e2)$.

3. *Either* $T(e1) < T(e2)$, *or* $T(e2) < T(e1)$ *or* $T(e1) \sim T(e2)$ *but no more than two of them holds.*

4. *Either* $T(e1) \lesssim T(e2)$, *or* $T(e2) \lesssim T(e1)$ *or both.*

5. *If* $T(e1) < T(e2)$ *and* $T(e2) \sim T(e3)$ *then* $T(e1) \lesssim T(e3)$.

6. *If* $T(e1) \sim T(e2)$ *and* $T(e2) < T(e3)$ *then* $T(e1) \lesssim T(e3)$.

7. *If* $\neg(T(e1) < T(e2))$ *then* $T(e2) \lesssim T(e1)$.

8. *If* $\neg(T(e1) < T(e2))$ *and* $\neg(T(e2) < T(e1))$ *then* $T(e1) \sim T(e2)$.

Proof.

1. (asymmetric) Deny, suppose both $T(e1) < T(e2)$ and $T(e2) < T(e1)$.

   Case1: $T(e1).site = T(e2).site$. Then $T(e1).local < T(e2).local$ and $T(e2).local < T(e1).local$. That contradicts to the asymmetric property of totally ordered local time.

   Case2: $T(e1).site \neq T(e2).site$. Then $((T(e1).global < T(e2).global - 1g_g) \wedge (T(e2).global < T(e1).global - 1g_g))$, which implies $(T(e1).global > T(e2).global + 1g_g)$. So we have: $T(e2).global + 1g_g < T(e2).global - 1g_g$. i.e., $1 < -1$, which is a contradiction.

2. (antisymmetric)

   Case 1: $T(e1).site = T(e2).site$ then

$$T(e1) \lesssim T(e2) \Rightarrow T(e1).local \leq T(e2).local$$

$$T(e2) \lesssim T(e3) \Rightarrow T(e2).local \leq T(e3).local$$

$$\Longrightarrow \quad T(e1).local = T(e2).local$$

$$\Longrightarrow \quad T(e1) = T(e2)$$

$$\Longrightarrow \quad T(e1) \sim T(e2)$$

Case 2: $T(e1).site \neq T(e2).site$ then

$$\begin{aligned} T(e1) \lesssim T(e2) \quad &\Rightarrow \quad T(e1) \sim T(e2) \text{ or } T(e1) < T(e2) \\ &\Rightarrow \quad |T(e1).global - T(e2).global| \leq 1g_g \\ &\text{or} \quad T(e1).global < T(e2).global - 1g_g \\ &\Rightarrow \quad T(e1).global \leq T(e2).global + 1g_g \\ &\Rightarrow \quad T(e1).global - T(e2).global \leq 1g_g \end{aligned}$$

Similarly we have:

$$\begin{aligned} T(e2) \lesssim T(e1) \quad &\Rightarrow \quad T(e2).global - T(e1).global \leq 1g_g \\ &\Longrightarrow \quad |T(e1).global - T(e2).global| \leq 1g_g \\ &\Longrightarrow \quad T(e1) \sim T(e2) \end{aligned}$$

3. In local and global time system, we have:
   $T(e1).local < T(e2).local$ or $T(e2).local < T(e1).local$ or $T(e1).local = T(e2).local$ but no more than two holds and $T(e1).global < T(e2).global - 1g_g$ or $T(e2).global < T(e1).global - 1g_g$ or $|T(e1).global - T(e2).global| \leq 1g_g$ but no more than two holds. So, we have $T(e1) < T(e2)$ or $T(e2) < T(e1)$ or $T(e1) \sim T(e2)$ but no more than two holds by definition.

4. By 3. either $T(e1) < T(e2)$ or $T(e2) < T(e1)$ or $T(e1) \sim T(e2)$ So, if $T(e1) < T(e2)$ then $T(e1) \lesssim T(e2)$; if $T(e2) < T(e1)$ then $T(e2) \lesssim T(e1)$. If $T(e1) \sim T(e2)$ then both $T(e1) \lesssim T(e2)$ and $T(e2) \lesssim T(e1)$.

5.

$$T(e1) < T(e2) \;\Rightarrow\; \begin{cases} T(e1).local & < & T(e2).local \text{ if } T(e1).site = T(e2).site. \\ & \Rightarrow & T(e1).global \leq T(e2).global \\ T(e1).global & < & T(e2).global \text{ if } T(e1).site \neq T(e2).site \end{cases}$$

$$T(e2) \sim T(e3) \;\Rightarrow\; \begin{cases} T(e2).local & = & T(e3).local \text{ if } T(e1).site = T(e2).site. \\ & \Rightarrow & T(e1).global = T(e2).global \\ |T(e2).global & - & T(e3).global| \leq 1g_g \\ & \Rightarrow & T(e3).global = T(e3).global - 1, \\ & & T(e3).global, T(e3).global + 1g_g \end{cases}$$

The worst case against the conclusion of $T(e1) \lesssim T(e3)$ is when $T(e1)$ is the biggest and $T(e3)$ is the smallest, which means

$$\begin{aligned} & T(e1).global \leq T(e2).global(\text{ same site }) \text{ and} \\ & T(e3).global = T(e2).global - 1g_g(\text{ different sites }) \\ \Longrightarrow\quad & T(e1).global \leq T(e3).global + 1g_g \\ \Longrightarrow\quad & T(e1) \lesssim T(e3) \end{aligned}$$

6. The proof is similar to above.

7. If $T(e1).site = T(e2).site$ then $\neg(T(e1) < T(e2))$ implies $T(e2).local \leq T(e1).local$. If $T(e2).local < T(e1).local$ then $T(e2) < T(e1)$ If $T(e2).local = T(e1).local$ then $T(e1) = T(e2)$, hence $T(e2) \sim T(e1)$. So we have $T(e2) \lesssim T(e1)$.
   If $T(e1).site \neq T(e2).site$ then $\neg(T(e1) < T(e2))$ implies $\neg(T(e2).global < T(e1).global - 1g_g)$, i.e., $T(e2).global \leq T(e1).global + 1$. If $T(e2).global < T(e1).global - 1$ then $T(e2) < T(e1)$ If $T(e1).global - 1g_g \leq T(e2).global \leq T(e1).global + 1$ then $T(e2) \sim T(e1)$. So we have $T(e2) \lesssim T(e1)$.

8. $\neg(T(e1) < T(e2)$ implies $T(e2) \lesssim T(e1)$ (by 7). similarly, $\neg(T(e2) < T(e1)$ implies $T(e1) \lesssim T(e2)$. So, by antisymmetricity, we have $T(e1) \sim T(e2)$.

□

CHAPTER 5
DISTRIBUTED COMPOSITE EVENT SEMANTICS

In this Chapter, the distributed composite time stamps and their ordering is defined based on the primitive time stamp definition from the previous chapter. The semantics of distributed composite operators of Sentinel is derived from the temporal relationship of the distributed time stamps.

5.1 Time Stamp of a Distributed Composite Event

The time stamp of composite events is different from that of primitive events in the sense that the time stamp of a composite event may be a set of time stamps instead of just one time stamp as is the case of primitive events. In the centralized systems, the time stamp of a composite event is defined as the latest time occurrence of the participating primitive events. Because of the partially ordered property of the global time, the "latest" is not uniquely defined. That is, there exists more than one global primitive time stamp that can be the "latest" and these "latest" ones are *concurrent* to each other. This gives rise to multiple time stamps. We define the "maximum" time stamps in a set of time stamps to be the ones (may not be unique) that are not less than any other time stamp in the set. Those time stamps from a set called the "max" of the given set of time stamps and the time stamps in the set of "max" is proven to be concurrent to each other.

Definition 5.1 (Set of maximum Time Stamps) *Given a set of time stamps ST, a time stamp $t \in ST$ is called a* maximum *of ST iff* $(\nexists t_1 \in ST, t < t_1)$ *The set of maximum time stamps in ST is defined as* $max(ST) = \{t \in ST : \ t \text{ is a maximum of } ST\}$

Theorem 5.1 *Given a set of primitive time stamps, ST, the time stamps in the max set max(ST) are concurrent to each other, i.e.,* $\forall t_1, t_2 \in max(ST), t_1 \sim t_2$ *where* $\sim$ *is the primitive concurrency relation defined earlier.*

Proof. Given a set of primitive time stamps ST, let $t_1, t_2 \in max(ST)$. Then t_1 is the *maximum* of ST, i.e., $(\nexists t \in ST, t_1 < t)$, so we have $\neg t_1 < t_2$. Similarly, we have $\neg(t_2 < t_1)$, which implies $t_1 \sim t_2$ by Proposition 4.3. $\square$

Definition 5.2 (Time Stamp of the Distributed Composite Event) *A time stamp of the distributed composite event e denoted* $T(e)$ *is a set of triples* $(site, global, local)$. *Each triple is a maximum of the set of time stamps of the constituent primitive event collected when the composite event occurs.*

We automatically have the property that the time stamps of a composite event are concurrent to each other by theorem 5.1. i.e., given $T(e)$ as the composite time stamp, then $\forall t_1, t_2 \in T(e), t_1 \sim t_2$ where $\sim$ is the primitive concurrency relation defined in the previous section.

Note that our definition of composite time stamps is different from [11]. Our definition is more precise in capturing the characteristics of the composite event time stamps, because in our definition, the concept of the "latest" is stressed and enforced and the concurrency within the time stamps is ensured by the theorem.

5.2 Partial Ordering on the Distributed Composite Time Stamps

Suppose $<_p$ is the strict partial order temporal relation on the set of distributed composite time stamps. In order for $<_p$ to make sense, we have the following requirements:

1. If $T(e1) <_p T(e2)$ then at least $\exists t_1 \in T(e1), \exists t_2 \in T(e2)$ such that $t_1 < t_2$ where $<$ is the strict partial order temporal relation on primitive time stamps defined in the previous section.

2. $<_p$ satisfies the irreflexity and transitivity, i.e, it is a well-defined strict partial ordering.

3. $<_p$ is the "least" restrictive in the sense that there does not exist any ordering $<_q$ which is more restrictive than $<_p$, ie, $\nexists <_q \forall T(e1), \forall T(e2)$. if $T(e1) <_p T(e2)$ then $T(e1) <_q T(e2)$.

A definition that satisfies requirements 1 and 2 is *valid* but may not be acceptable. For example, let us define: $T(e1) <_p T(e2)$ iff $\forall t1 \in T(e1), \forall t_2 \in T(e2), t1.global < t2.global - 10g_z$. The above definition satisfies 1 and 2 but is not acceptable because it is too "restrictive" in the sense that there are a lot of time stamps that cannot be compared using the definition. It requires that all the constituent primitive time stamps of T(e2) to be at least $10g_g$ greater than all the ones in T(e1), which is too restrictive and does not make sense. The third requirement is added to to ensure that most pairs of time stamps can be compared.

Let $<_p$ be a partial ordering that satisfies the above requirements, we have:

$$T(e1) <_p T(e2) \Rightarrow \exists t_1 \in T(e1), \exists t_2 \in T(e2), t_1 < t_2$$

$$T(e2) <_p T(e3) \Rightarrow \exists t_2 \in T(e2), \exists t_3 \in T(e3), t_2 < t_3$$

by our requirement 1. Also, the transitivity needs to be satisfied.

$$T(e1) <_p T(e2), T(e2) <_p T(e3) \Rightarrow T(e1) <_p T(e3)$$

for which we need:

$$\exists t_1 \in T(e1), \exists t_2 \in T(e2), t_1 < t_2 \tag{5.1}$$

$$\exists t_2 \in T(e2), \exists t_3 \in T(e3), t_2 < t_3 \tag{5.2}$$

$$\Rightarrow \quad \exists t_1 \in T(e1), \exists t_3 \in T(e3), t_1 < t_3 \tag{5.3}$$

Because the value of t_2 in (5.1) may be different from the value of t_2 in (5.2) (due to the existantial quantification), we can not conclude that the third equation holds.

We need at least one of the $\exists$ to be $\forall$ in the above three equations. If not, there could always exist cases when the transitivity property does not hold, i.e, the transitivity may not be guaranteed. So, a definition of

$$T(e1) <_p T(e2) \quad \text{iff} \quad (\forall t_2 \in T(e2), \exists t_1 \in T(e1))(t_1 < t_2) \text{ or}$$
$$T(e1) <_q T(e2) \quad \text{iff} \quad (\forall t_1 \in T(e1), \exists t_2 \in T(e2))(t_1 < t_2)$$

becomes the only two valid definitions with the "least restrictive" strict ordering definition that satisfies the requirements specified earlier. Any other valid definitions will need at least one $\forall$ and one $\exists$ to ensure the transitivity which means that are more restrictive. If we denote:

$$T(e1) >_p T(e2) \quad \text{iff} \quad (\forall t_2 \in T(e2), \exists t_1 \in T(e1))(t_1 > t_2) \text{ and}$$
$$T(e1) >_q T(e2) \quad \text{iff} \quad (\forall t_1 \in T(e1), \exists t_2 \in T(e2))(t_1 > t_2)$$

then

$$T(e1) <_p T(e2) \Leftrightarrow T(e2) >_q T(e1) \text{ and}$$
$$T(e1) <_q T(e2) \Leftrightarrow T(e2) >_p T(e1).$$

So, $(<_p, >_q)$ and $(<_q, >_p)$ are two *dual* pairs of ordering relation satisfying the above three requirements. In this paper, $(<_p, >_q)$ is chosen to be our definition of strict ordering of the composite time stamps.

Notice that the definition of

$$T(e1) <_{p1} T(e2) \text{ iff } (\exists t_1 \in T(e1), \exists t_2 \in T(e2))(t_1 < t_2)$$

is not valid since it is not transitive.

There are some other interesting definitions that are valid but not the "least" restrictive:

1. $T(e1) <_{p2} T(e2)$ iff ($\forall t_1 \in T(e1), \forall t_2 \in T(e2))(t_1 < t_2)$.

   This one is valid but is more restrictive than $<_p$. Here is an example: $T(e1) = \{(site1, 8, 80), (site2, 7, 70)\}$ and $T(e2) = \{(site3, 9, 90)\}$ satisfies $<_p$ relation but not the $<_{p2}$ defined above.

2. Let mt_1 be the time stamp in $T(e1)$ with minimum global time. $T(e1) <_{p3} T(e2)$ iff $\forall t_2 \in T(e2), (mt_1 < t_2)$.

   This one is also valid but is more restrictive than $<_p$. Here is an example: $T(e1) = \{(site1, 8, 80), (site2, 7, 70)\}$ and $T(e2) = \{(site1, 8, 81), (site2, 7, 71)\}$ satisfies $<_p$ relation but not the $<_{p3}$ defined above, since $(site1, 8, 81)$ in T(e2) is $\not<_p$ (site2, 7,70) in $T(e1)$ which has the minimal global time .

Based on the above analysis, we have the following definition:

Definition 5.3 *Let $T(e1)$ and $T(e2)$ be the two distributed composite event time stamps. Then the temporal relationship between $T(e1)$ and $T(e2)$ is defined as the following:*

1. *concurrency:*

$$
\begin{aligned}
T(e1) \sim T(e2) \quad & \textit{iff} \quad (\forall t_1 \in T(e1) \forall t_2 \in T(e2)) \\
& \qquad (t_1.site = t_2.site \wedge t_1.local = t_2.local) \\
& \vee \quad (t_1.site \neq t_2.site \wedge \mid t_1.local - t_2.local \mid < 2g_g) \\
OR \quad & \textit{iff} \quad (\forall t_1 \in T(e1) \forall t_2 \in T(e2)) \\
& \qquad t_1 \sim t_2 \\
OR \quad & \textit{iff} \quad \forall t_2 \in T(e2), t_2 \in C(T(e1)), \\
& \qquad \textit{where } C(T(e1)) \textit{ is the concurrent set of } T(e1).
\end{aligned}
$$

(It's easy to see the equivalence of the three concurrency definition above).

2. happen before :

$$\begin{aligned} T(e1) < T(e2) \quad & \textit{iff} \quad (\forall t_1 \in T(e1), \forall t_2 \in T(e2))(t_1 \leq t_2) \\ & \qquad \wedge (\forall t_2 \in T(e2), t_2 \notin C(T(e1))) \\ OR \quad & \textit{iff} \quad (\forall t_2 \in T(e2), \exists t_1 \in T(e1))(t_1 \leq t_2) \end{aligned}$$

Proof of the equivalence of the above two definitions of happen before *:*

$$\begin{aligned} T(e1) < T(e2) \quad & \textit{iff} \quad (\forall t_1 \in T(e1), \forall t_2 \in T(e2))(t_1 \leq t_2) \\ & \qquad \wedge (\forall t_2 \in T(e2), t_2 \notin C(T(e1))) \\ & \textit{iff} \quad (\forall t_1 \in T(e1), \forall t_2 \in T(e2))(t_1 \leq t_2) \\ & \qquad \wedge (\forall t_2 \in T(e2), \exists t_1 \in T(e1))(t_1 \not\sim t_2) \\ & \textit{iff} \quad (\forall t_1 \in T(e1), \forall t_2 \in T(e2))(t_1 \leq t_2) \\ & \qquad \wedge (\forall t_2 \in T(e2), \exists t_1 \in T(e1))(t_1 < t_2) \\ & \qquad (t_1 \textit{ and } t_2 \textit{ are primitive and if } t_1 \leq t_2 \textit{ and } t_1 \not\sim t_2 \textit{ then } t_1 < t_2) \\ & \textit{iff} \quad (\forall t_2 \in T(e2), \exists t_1 \in T(e1))(t_1 < t_2) \end{aligned}$$

3. incomparable:

$$T(e1) \bowtie T(e2) \textit{ iff } \neg((T(e1) < T(e2)) \vee (T(e1) > T(e2)) \vee (T(e1) \sim T(e2)))$$

Theorem 5.2 The $<$ defined above is irreflexive *and* transitive *hence is a well-defined strict partial ordering on the set of all distributed composite event time stamps.*

Proof. Proof of irreflexiveness, i.e., $\neg(T(e) < T(e)), \forall T(e)$.

Deny, suppose $T(e) < T(e)$, then by definition, $\exists t_1 \in T(e), \forall t \in T(e), t < t_1$ which includes t_1. This implies $t_1 < t_1$ which contradicts to the irreflexiveness of $<$ of the primitive time stamps.

Proof of transitivity: i.e., if $T(e1) < T(e2)$, and $T(e2) < T(e3)$ then $T(e2) < T(e3)$.

$$\begin{aligned} T(e1) < T(e2) &\Leftrightarrow (\forall t_2 \in T(e2), \exists t_1 \in T(e1))(t_1 < t_2) \\ T(e2) < T(e3) &\Leftrightarrow (\forall t_3 \in T(e3), \exists t_2 \in T(e2))(t_2 < t_3) \\ &\Longrightarrow \quad (\forall t_3 \in T(e3), \exists t_1 \in T(e1))(t_1 < t_3) \end{aligned}$$

So, we have $T(e1) < T(e3)$. $\Box$

A semantics of $\lesssim$ can be defined as follow. Interesting enough, this definition can be proved to be consistent with the definition of primitive $\lesssim$ very easily.

Definition 5.4 (weakened-less-than-or-equal-to)

$$T(e1) \lesssim T(e2) \textit{ iff } (\forall t_1 \in T(e1) \forall t_2 \in T(e2))(t_1 \lesssim t_2).$$

Theorem 5.3 $T(e1) \lesssim T(e2)$ *iff* $T(e1) \sim T(e2)$ *or* $T(e1) < T(e2)$.

Proof. $\Longrightarrow$:

Suppose $T(e1) \lesssim T(e2)$ want to show:$(T(e1) \sim T(e2)) \vee (T(e1) < T(e2))$

$$\begin{aligned} T(e1) \lesssim T(e2) \Rightarrow & \quad \forall t_1 \in T(e1), \forall t_2 \in T(e2), t_1 \lesssim t_2 \\ \Rightarrow & \quad \forall t_1 \in T(e1), \forall t_2 \in T(e2), t_1 < t_2 \text{ or } t_1 \sim t_2 \\ \Rightarrow & \quad (\forall t_1 \in T(e1), \forall t_2 \in T(e2), t_1 < t_2) \text{ or } (\forall t_1 \in T(e1), \forall t_2 \in T(e2), t_1 \sim t_2) \\ \Rightarrow & \quad (\forall t_2 \in T(e2), \exists t_1 \in T(e1), t_1 < t_2) \text{ or } (\forall t_1 \in T(e1), \forall t_2 \in T(e2), t_1 \sim t_2) \\ \Rightarrow & \quad (T(e1) < T(e2)) \text{ or } (T(e1) \sim T(e1)). \end{aligned}$$

$\Longleftarrow$: Suppose $(T(e1) \sim T(e2)) \vee (T(e1) < T(e2))$ want to show: $T(e1) \lesssim T(e2)$

$$(T(e1) < T(e2)) \vee (T(e1) \sim T(e2)) \quad \Rightarrow \forall t_2 \in T(e2) \exists t_{11} \in T(e1) t_{11} < t_2 \qquad (5.4)$$

$$\text{or } \forall t_1 \in T(e1) \forall t_2 \in T(e2) t_1 \sim t_2 \qquad (5.5)$$

By definition of distributed time stamps, every two primitive time stamps are concurrent to each other, we have: $\forall t_1 \in T(e1) t_1 \sim t_{11}$. Combined with equation 5.4,

we have: $\forall t_2 \in T(e2), \forall t_1 \in T(e1), \exists t_{11} \in T(e1) t_1 \sim t_{11}$ and $t_{11} < t_2$. Which implies: $\forall t_2 \in T(e2), \forall t_1 \in T(e1), \exists t_{11} \in T(e1) t_1 \stackrel{<}{\sim} t_2$ by proposition 4.3. i.e. $\forall t_2 \in T(e2), \forall t_1 \in T(e1), t_1 \stackrel{<}{\sim} t_2$. So, we have: $T(e1) \stackrel{<}{\sim} T(e2)$. From 5.5 we also have: $T(e1) \stackrel{<}{\sim} T(e2)$. So, $T(e1) \stackrel{<}{\sim} T(e2)$ holds. $\square$

Our definition is different from the [11]. The "happen before" definition in [11] is not transitive even though the dissertation indicates otherwise. Here is counter example: Let $T(e1) = \{(site1, 8, 80)\}$, $T(e2) = \{(site1, 9, 90), (site2, 8, 80)\}$, and $T(e3) = \{(site2, 9, 90)\}$. It's easy to see that $T(e1) < T(e2), T(e2) < T(e3)$, by their definition, but $T(e1) \sim T(e3)$. So, their definition of $<$ is *not* a well-defined strict partial order temporal relation. On the other hand our definition depends only on the definition of the distributed primitive event time stamps. The concepts of "sites", "global time" and "local time" are embedded in the definition of primitive event time stamps and becomes transparent to composite events. One other obvious advantage is that our definition is a mathematically well-defined ordering and hence ensure to be the "best" by mathematical reasoning. A semantics of open and closed intervals, needed for the complexity of Sentinel composite event operator can be derived easily from the $<$ ordering. Also, the temporal relation between any two distributed time stamps is very easy to visualize via our graph presentation show below.

The time stamps in a distributed system can be represented as a two dimensional grid with x-axis being the global time embedded with the local time and y-axis being the sites in the distributed system. The following is an example indicating the composite time stamp and its $\sim$, $<, \stackrel{<}{\sim}$ area. Let the global distributed composite event be $T(e) = \{(Site3, 8, 81), (Site6, 7, 72)\}$. For any composite time stamp $T(e1)$,

1. $T(e1) \sim T(e)$ iff $T(e1)$ lies between Line2 and Line3;
2. $T(e1) < T(e)$ iff $T(e1)$ lies before Line1 ;
3. $T(e) < T(e1)$ iff $T(e1)$ lies after Line4;

4. $T(e1) \lesssim T(e)$ iff $T(e1)$ lies before Line3 ; and

5. $T(e) \lesssim T(e1)$ iff $T(e1)$ lies after Line2;

See Figure 5.1. A time stamp that crosses those lines indicates a incomparable situation.

Analogously, the notation of *Open* and *Closed* interval can defined as follows:

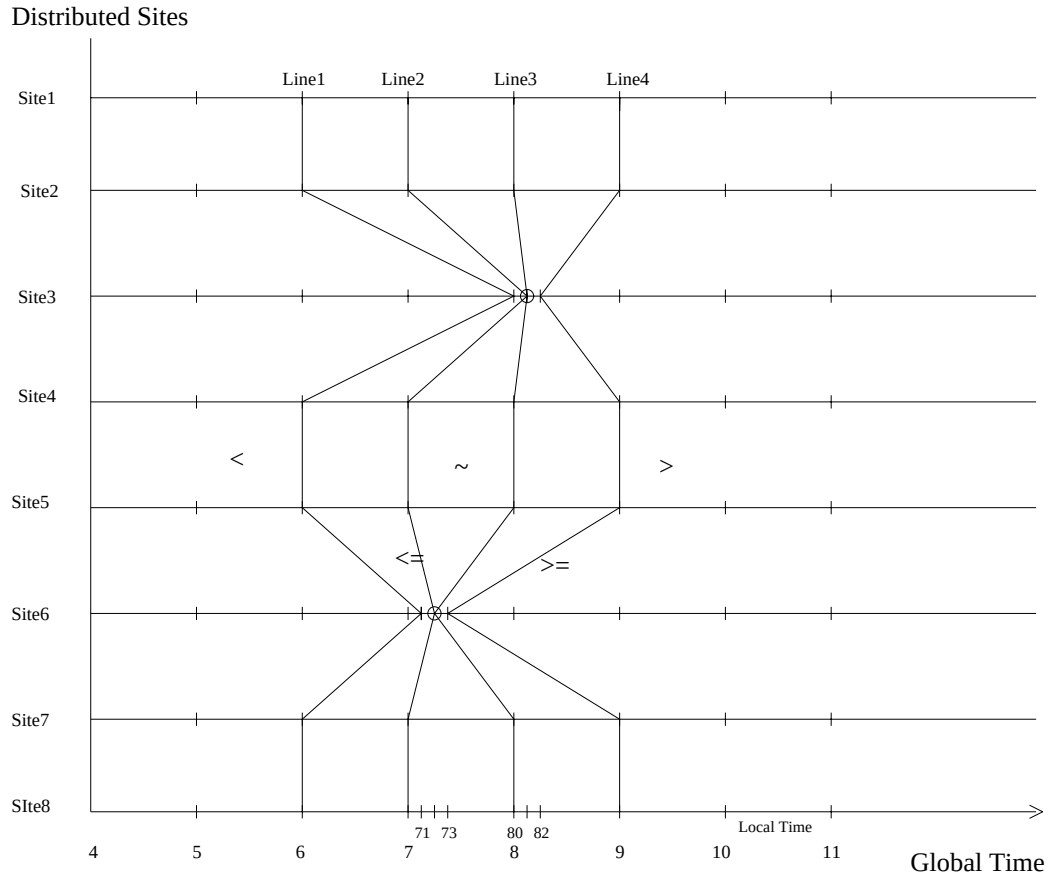

Figure 5.1. An example of composite event and its temporal area.

Definition 5.5 (Open Interval of Composite Time stamps in Distributed Systems) *Let T(e1), T(e2) be the time stamps of global composite event* $e1$ *and* $e2$. *An event e with time stamp T(e) is said to be in the open interval formed by* $T(e1)$ *and* $T(e2)$ *with* $T(e1) < T(e2)$ *denoted*

$$T(e) \in \tilde{(}\ T(e1), T(e2)\ \tilde{)} \quad \textit{iff}\ T(e1) < T(e) < T(e2).$$

Definition 5.6 (Closed Interval of Composite Time stamps in Distributed Systems) *Let T(e1), T(e2) be the time stamps of global composite event* $e1$ *and* $e2$. *A time stamp T(e) is said to be in the closed interval formed by* $T(e1)$ *and* $T(e2)$ *(requires* $T(e1) \lesssim T(e2)$ *) denoted* $T(e) \in \tilde{[}\ T(e1), T(e2)\ \tilde{]}$ *iff* $T(e1) \lesssim T(e) \lesssim T(e2)$.

The following are some properties derived from the definition which may not so easy to visualize.

Proposition 5.1 *Let* $T(e1), T(e2)$ *and* $T(e3)$ *be any three distributed composite time stamps.*

1. *(asymmetric) If* $T(e1) < T(e2)$, *then* $\neg(T(e2) < T(e1))$.
2. *(antisymmetric) If* $T(e1) \lesssim T(e2)$ *and* $T(e2) \lesssim T(e1)$ *then* $T(e1) \sim T(e2)$.
3. *If* $T(e1) \sim T(e2)$ *and* $T(e2) < T(e3)$ *then* $T(e1) \lesssim T(e3)$.
4. *Let* $T(e1), T(e2)$ *be two global event time stamps such that* $T(e1) \sim T(e2)$ *then* $\forall t_1 \in T(e1), \forall t_2 \in T(e2)$ *we have* $|t_1.global - t_2.global| \leq 1g_g$.
5. *Let* $T(e1), T(e2)$ *be two global event time stamps such that* $T(e1) \bowtie T(e2)$ *then* $\forall t_1 \in T(e1), \forall t_2 \in T(e2)$ *we have* $|t_1.global - t_2.global| \leq 2g_g$.

Proof.

1. (asymmetric): Deny, suppose $\exists T(e1) and T(e2)$ satisfies both $T(e1) < T(e2)$ and $T(e2) < T(e1)$, then

$$
\begin{aligned}
T(e1) < T(e2) &\Rightarrow \forall t_2 \in T(e2) \exists t_{11} \in T(e1), t_{11} < t_2 \\
T(e2) < T(e1) &\Rightarrow \forall t_1 \in T(e1) \exists t_{21} \in T(e2), t_{21} < t_1 \\
&\Rightarrow \forall t_2 \in T(e2) \exists t_{11} \in T(e1), t_{11} < t_2, \\
&\qquad \text{let } t_1 = t_{11}, \text{ then } \exists t_{22} \in T(e2), t_{22} < t_{11} \\
&\Rightarrow \forall t_2 \in T(e2) \exists t_{11} \in T(e1), \exists t_{22} \in T(e2), t_{11} < t_2, \text{ and } t_{22} < t_{11} \\
&\Rightarrow \forall t_2 \in T(e2) \exists t_{22} \in T(e2), t_{22} < t_2
\end{aligned}
$$

This contradicts to the definition of composite time stamps, i.e., for any two time stamps in T(e2), the must be concurrent to each other.

2. (antisymmetric):

$$\begin{aligned}
T(e1) \lesssim T(e2) &\Rightarrow \forall t_1 \in T(e1), \forall t_2 \in T(e2), t_1 \lesssim t_2. \\
T(e2) \lesssim T(e1) &\Rightarrow \forall t_2 \in T(e2), \forall t_1 \in T(e1), t_2 \lesssim t_1. \\
&\Rightarrow \forall t_1 \in T(e1), \forall t_2 \in T(e2), t_1 \lesssim t_2 \text{ and } t_2 \lesssim t_1. \\
&\Rightarrow \forall t_1 \in T(e1), \forall t_2 \in T(e2), t_1 \sim t_2 \\
&\Rightarrow T(e1) \sim T(e2).
\end{aligned}$$

3.

$$\begin{aligned}
T(e1) \sim T(e2) &\Rightarrow \forall t_1 \in T(e1), \forall t_2 \in T(e2), t_1 \sim t_2. \\
T(e2) < T(e3) &\Rightarrow \forall t_3 \in T(e3), \exists t_2 \in T(e2), t_2 < t_3. \\
&\Rightarrow \forall t_1 \in T(e1), \forall t_3 \in T(e3), \exists t_2 \in T(e2), t_1 \sim t_2 \text{ and } t_2 < t_3 \\
&\Rightarrow \forall t_1 \in T(e1), \forall t_3 \in T(e3), t_1 \lesssim t_3 \\
&\Rightarrow T(e1) \lesssim T(e3).
\end{aligned}$$

4. Deny, suppose $\exists t_1 \in T(e1), t_2 \in T(e2), |t_1.global - t_2.global| > 1$.. Without lost of generality assuming $t_1.global < t_2.global$, we have:

   $t_1.global < t_2.global - 1$, i.e. $t_1 < t_2$ disregard the sites of t_1 and t_2. That contradicts the definition of $T(e1) \sim T(e2)$

5. Deny, suppose $\exists t_1 \in T(e1), t_2 \in T(e2), |t_1.global - t_2.global| > 2$.

   Without lost of generality assuming $t_1.global < t_2.global$,

   we have: $t_1.global < t_2.global - 2$.

   $\forall t_{20} \in T(2)$, we have: $t_2 \sim t_{20}$, that is, $t_2.global = t_{20}.global - 1$, $t_{20}.global$,

and $t_{20}.global + 1$.

In any cases, we have $t_1.global < t_2.global - 1$, $\forall t_{20}$ in $T(e2)$.

So, we have $\forall t_{20} \in T(e2), \exists t_1 \in T(e1), t_1 < t_{20}$ disregard the sites of t_1 and t_{20}, i.e., $T(e1) < T(e2)$ by definition. That contradicts to the definition of $T(e1) \bowtie T(e2)$.

□

The following is a example demonstrating the temporal relationship of global event time stamps.

Example 5.1 *Let k, l, m be a set of physical clocks in different sites of the Distributed system with granularity $g = 1/100s$. Let z be the reference clock with granularity $g_z = 1/1000s$. Assume the physical clocks are synchronized with precision $\Pi < 1/10s$ and the global granularity is chosen to be $g_g = 1/10s$. Let $T(e1), T(e2), T(e3), T(e4), T(e5)$ be the sets of time stamps of the global composite events $e1, e2, e3, e4, e5$:*

$$
\begin{aligned}
T(e1) = \ & \{ \quad (k, 2399154827, 23991548276), \\
& \quad\ (m, 2399154827, 23991548277)\} \\
T(e2) = \ & \{ \quad (l, 2399154827, 23991548276), \\
& \quad\ (k, 2399154827, 23991548277)\} \\
T(e3) = \ & \{ \quad (m, 2399154827, 23991548276), \\
& \quad\ (l, 2399154827, 23991548277)\} \\
T(e4) = \ & \{ \quad (k, 2399154828, 23991548288), \\
& \quad\ (l, 2399154827, 23991548277)\} \\
T(e5) = \ & \{ \quad (k, 2399154829, 23991548298), \\
& \quad\ (l, 2399154828, 23991548287)\}
\end{aligned}
$$

Then, $T(e1) \bowtie T(e2) \bowtie T(e3)$, $T(e4) \sim T(e3)$ *and* $T(e3) < T(e5)$

5.3 Distributed Composite Time Stamp Operation

In centralized systems, when a composite event is detected, a new time stamp (t_occ) indicating the latest time stamp (that made the composite event occur) along with the event name, and event parameters are propagated to the parent nodes (if any). Similarly, in distributed systems, when the composite event is detected, a set of time stamps indicating the "latest" time stamps need to be generated and sent to the parent node with the event type and parameters. This procedure of generating a set of "latest" time stamps is defined as a *Max* operator. The resulting composite time stamp generated by the Max operator also needs to satisfy the definition of the distributed composite time stamps.

The Max operator can be specified by different procedures depending on the relationship between two time stamps. Before defining the *Max* of two sets of time stamps, the *joining* of concurrent and incomparable time stamps are defined as follows:

Definition 5.7 (Joining procedure for concurrent time stamps) Let T(e1) and T(e2) be two global time stamps such that $T(e1) \sim T(e2)$. *Then the joining time stamp of T(e1) and T(e2) denoted* $T(e1) \cup T(e2)$ *is defined as:*

$$T(e1) \cup T(e2) = \{ts | ts \in T(e1) \text{ or } ts \in T(e2)\}.$$

The joining of $T(e1)$ and $T(e2)$ joins the corresponding component time stamps and eliminates the duplicates which is exactly the same as the set union operation in mathematics. The concurrency property is ensured by proposition 5.1.

Definition 5.8 (Joining procedure on incomparable time stamps) Let T(e1) and T(e2) be two global time stamps such that $T(e1) \bowtie T(e2)$. *Then the joining time stamp of*

T(e1) and T(e2) denoted $T(e1) \cup T(e2)$ *is defined as:*

$$\begin{aligned} T(e1) \cup T(e2) &= \{ts | ts \in T(e1) \text{ such that } \forall ts_2 \in T(e2), \neg(ts < ts_2)\} \\ &\cup \\ &\{ts | ts \in T(e2) \text{ such that } \forall ts_1 \in T(e1), \neg(ts < ts_1)\}. \end{aligned}$$

The "joining" of $T(e1) \bowtie T(e2)$ is to keep the "latest" information of the two sets of time stamps. The result composite time stamps after the join operation can be proved to satisfy the definition of the composite time stamps. The above two *joining* procedures are conceptually same as the joining in [11]. Our definition is more precise and easy to understand.

Definition 5.9 (the Maximum of time stamps) Let $T(e1)$ *and* $T(e2)$ *be the two sets of the time stamps of the distributed composite events, then the maximum of* $T(e1)$ *and* $T(e2)$ *denoted* $Max(T(e1), T(e2))$ *is defined as follows:*

$$Max(T(e1), T(e2)) = \begin{cases} T(e1) & \text{if } T(e1) > T(e2) \\ T(e2) & \text{if } T(e2) > T(e1) \\ T(e1) \cup T(e2) & \text{if } T(e1) \text{ and } T(e2) \\ & \text{are concurrent or incomparable} \end{cases}$$

Theorem 5.4 If $T(e) = Max(T(e1), T(e2))$ *where* $T(e1)$ *and* $T(e2)$ *are any time stamps then* $T(e)$ *is a distributed composite time stamp with the primitive components from* $T(e1)$ *and* $T(e2)$*, i.e., let* $ST = T(e1) \cup T(e2)$ *where* $\cup$ *is the ordinary mathematics union operator, then* $T(e) = max(ST)$.

Proof. If $T(e1) < T(e2)$ or $T(e2) < T(e1)$, it is trivial to see the result holds. If $T(e1) \sim T(e2)$, the result holds by proposition 5.1(4). If $T(e1) \bowtie T(e2)$,then the result holds by definition of $T(e1) \cup T(e2)$. Otherwise, if $\exists t_1, t_2, t_1 < t_2$, then t_1, t_2 can not be in the same set of time stamp (otherwise it contradicts the concurrence property within the time stamp). Suppose $t_1 \in T(e1)$ and $t_2 \in T(e2)$, then $\exists t_1 \in T(e1), t_2 \in T(e2), t_1 < t_2$ which contradicts the definition of the $\cup$ operator in Definition 5.8. $\square$

Example 5.2 *Let the assumptions and* $T(e1), T(e2), T(e3), T(e4)$ *and* $T(e5)$ *be the same as previous example. Then*

$$\begin{aligned} T(e1) \cup T(e2) = \quad & \{ \quad (k, 2399154827, 23991548276), \\ & \quad\ (m, 2399154827, 23991548277), \\ & \quad\ (l, 2399154827, 23991548276)\} \\ T(e3) \cup T(e4) = \quad & \{ \quad (m, 2399154827, 23991548276), \\ & \quad\ (l, 2399154827, 23991548277), \\ & \quad\ (k, 2399154828, 23991548288)\} \\ T(e3) \cup T(e5) = \quad & \{ \quad (k, 2399154829, 23991548298), \\ & \quad\ (l, 2399154828, 23991548287)\} \end{aligned}$$

5.4 The Semantics of Distributed Composite Event Operators

A distributed event E (either primitive or composite is a function from the time stamp domain onto the boolean values, True or False.

$$E : TS \rightarrow \{True, False\}$$

given by:

$$E(ts) = \begin{cases} T(rue) & \text{if an event of type E occurs with time stamp ts} \\ F(alse) & \text{otherwise} \end{cases}$$

Here the time stamp ts could be a set of time stamps formed by the Max operator collecting from the corresponding primitive event time stamps if E is a global composite event. Based on the time stamp set operation defined in last section, the semantics of the distributed composite event operators can be defined as follows:

1. OR(Δ): Disjunction of two events. Denoted $E_1 \Delta E_2$. Formally,

$$\begin{aligned} (E_1 \Delta E_2)(ts) \quad = \quad & (E_1(ts) \wedge (\sim E_2(ts))) \\ & \vee (E_2(ts) \wedge (\sim E_1(ts))) \\ & \vee (\exists ts1, ts2)(E_2(ts1) \wedge E_2(ts2)) \wedge (ts = Max(ts1, ts2)) \end{aligned}$$

2. AND(∇): Conjunction of two events E_1 and E_2, denoted $E_1 \nabla E_2$. Formally,

$$(E_1 \nabla E_2)(ts) = (\exists ts1, ts2)(E_1(ts1) \wedge E_2(ts2)) \wedge (ts = Max(ts1, ts2))$$

3. ANY: The conjunction of events, denoted $ANY(m, E_1, E_2, ...E_n)$. Formally,

$$\begin{aligned} ANY(m, E_1, E_2, ..., E_n)(ts) \quad = \quad & (\exists ts_1, ts_2, ..., ts_m) \\ & (E_i(ts_1) \wedge E_j(ts_2) \wedge \ldots \wedge E_l(ts_m))) \\ & \wedge (1 \leq i, j, \ldots, l \leq n) \\ & \wedge (i \neq j \neq \ldots \neq l) \\ & \wedge (ts = max(ts_1, ts_2, \ldots, ts_m)) \end{aligned}$$

4. SEQ(;): The sequence of two events $E1$ and $E2$, denoted $E1; E2$. Formally,

$$(E1; E2)(ts) = (\exists ts_1)(E_1(ts_1) \wedge E_2(ts)) \wedge (ts_1 < ts)$$

5. Aperiodic Operators (A, A^*):

   Non-cumulative variant of an aperiodic event, denoted A(E_1, E_2, E_3). Formally,

$$\begin{aligned} A(E_1, E_2, E_3)(ts) \quad = \quad & (\exists ts)(\forall ts_2, ts_1 \leq ts_2 < ts) \\ & (E_1(ts_1) \wedge \sim E_3(ts_2) \wedge E_2(ts)) \end{aligned}$$

   The accumulative version, denoted $A^*(E_1,\ E_2,\ E_3)$. Formally,

$$A^*(E_1, E_2, E_3)(ts) \quad = \quad (\exists ts_1, ts_1 < ts)(E_1(ts_1) \wedge E_3(ts))$$

6. **Periodic Event Operators (P, P*):** The non-cumulative version of periodic event, denoted $P(E_1, TI[: parameters], E_3)$. Formally,

$$\begin{aligned} P(E_1, TI[: parameters], E_3)(ts) \quad = \quad & (\exists ts_1) \\ & (\forall ts_2 \in [ts_1, ts], ts = ts_1 + i * TI \text{ for some i }) \\ & (E_1(ts_1) \wedge \sim E_3(ts_2)) \end{aligned}$$

   The cumulative version of periodic event, denoted $P^*(E_1, TI[: parameters], E_3)$. Formally,

$$P^*(E_1, TI[: parameters], E_3) = (\exists ts_1)(ts > ts_1)(E_1(ts_1) \wedge E_3(ts))$$

where TI is a time specification.

7. **Not ($\neg$):** The not operator, denoted $\neg(E_2)[E_1,\ E_3]$. Formally,

$$\neg(E_2)[E_1, E_3](ts) \quad = \quad (\exists ts_1)(\forall ts_2 \in [ts_1, ts])(E_1(ts_1) \wedge \sim E_2(ts_2) \wedge E_3(ts))$$

CHAPTER 6
DISTRIBUTED COMPOSITE EVENT EVALUATION AND IMPLEMENTATION

Because of the message delay in distributed environments, the arrival order of the events at GED may not correspond to the order of occurrence of the events. The delays may be due to delays of the network, disruptions of the sites where the events occur or disruption of the network. There are two possible policies of distributed composite event evaluation discussed in [11]. One is *Asynchronous Evaluation* which ignores the fact that there may be delayed events and evaluates global event tree as soon as suitable events arrive at an observer site, and the other is *Synchronous Evaluation* which waits for delayed events and evaluates global event tree only if all relevant events have arrived at an observer site.

6.1 Synchronous Evaluation

Definition 6.1 (Synchronous Evaluation) A distributed composite event node is evaluated synchronously*, if each node is evaluated on the arrival of an event occurrence from a child node provided that all event occurrences from other child nodes with smaller time stamps have arrived.*

Synchronous evaluation assumes the implementation of FIFO network delivery, that is, the messages originating at any one site are delivered at any other site in the order they were generated. FIFO message delivery can be easily achieved by TCP/IP. Synchronous evaluation means that the evaluation of a node is *delayed* until all relevant events with smaller time stamps have arrived at the global event detector. The main advantage of synchronous evaluation is that it guarantees that the events are

detected in parameter contexts. It is suitable for applications requiring a high degree of consistency and reliability. The main disadvantage is that the evaluation can be blocked for a long time, if there are site failures or networking congestion. An other draw back is that the storage requirement can be very large depending on the characteristics of arriving events.

6.2 Asynchronous Evaluation

Definition 6.2 (Asynchronous Evaluation) A distributed composite event node is evaluated asynchronously*, if each node is evaluated instantly on the arrival of an event occurrence form the child node.*

Asynchronous evaluation means that the composite nodes are evaluated irrespective of networking delays or failures. When events arrive at a node, there may be other events with smaller time stamps which have not yet arrived. But the composite event is evaluated instantly. This means that the events are not necessarily evaluated in the order of their occurrence. The main advantage of asynchronous evaluation is the fast response time which is suitable for real time applications. The main draw back is that it does not guarantee detection of all events in parameter context. The context is enforced on current available events.

6.3 Evaluation Policies and Context Constrains

Even though the parameter context constraint is not globally guaranteed in asynchronous evaluation, we still want to flush out the unwanted events based on the idea of context constraints in a centralized environment. The context constraint has to be loosened for the distributed case and the evaluation is based on the *moment* when an event occurs. Before describing the algorithms based on the evaluation policies, we need have a review about the context constraints.

The context constraint are motivated by a careful analysis of several classes of applications [18] and are believed to be useful for a wide range of applications.

- **Recent:** In this context, only the most recent occurrence of the initiator for any event that has started the detection of that event is used. When an event occurs, the event is detected and all the occurrences of events that cannot be the initiators of that event in the future are deleted (or flushed). In this context, *not all occurrences* of a constituent event will be used in detecting a composite event. Furthermore, an initiator of an event (primitive or composite) will continue to initiate new event occurrences until a new initiator occurs.

- **Chronicle:** In this context, for an event occurrence, the initiator, terminator pair is unique. The oldest initiator is paired with the oldest terminator for each event (i.e., in chronological order of occurrence). When an event X is detected, its parameters are computed by using the oldest initiator and the oldest terminator of E. However, the constituent events of an event X cannot occur as a constituent event in any other occurrence of a composite event.

- **Continuous:** In this context, each initiator of an event starts the detection of that event. A terminator event occurrence may detect one or more occurrences of the same event. The initiator and the terminator are discarded after an event is detected. This context is especially useful for tracking trends of interest on a sliding time point governed by the initiator event. In this context, an initiator will be used *at least once* for detecting that event.

  There is a subtle difference between the chronicle and the continuous contexts. In the former, pairing of the initiator is with a unique terminator of the event whereas in the latter multiple initiators are paired with a single terminator of that event.

- **Cumulative:** In this context, all occurrences of an event type are accumulated as instances of that event until the event is detected. Whenever an event is

detected, all the occurrences that are used for detecting that event are deleted. Unlike the continuous context, an event occurrence does not participate in two distinct occurrences of the same event in the cumulative context.

One advantage of the Synchronous evaluation policy is that it guarantees that the event detection satisfies the parameter context constraint. The following is an example algorithm of SEQ operator with Recent context:

```
SEQ(E1, E2):
    if e1 signaled
      if E2 != empty and if ( exists e2 in E2, e1 < e2 )
      //all related e1 has arrived
      //all e2 with e2<e1 can be evaluated now.
        let SeqE2 =  e2 in E2 such that e2 < e1
        if SeqE2!= empty
          for all e2 in SeqE2
          let RE1 =  e1 in E1 such that e1 < e2
          //RE1 is the set of most recent e1<e2
              for all e1 in RE1
                  propagate < e1, e2 > to parents
        delete e2 in SeqSet form E2
      insert e1
    if e2 signaled
      if E1 != empty and if exists e1 in E1 such that e2 < e1
      //all relevant e1 has arrived, e2 can be evaluated now
      let RE1 =  e1 in E1 such that e1 < e2 and
      //RE1 is the set of most recent e1< e2.
        for all e1 in RE1
```

propagate < e1, e2 > to parents

else insert e2

delete all e1 with maxming(e1)<maxming(e2)-9

Here $maxming(T) = maxg(T) + ming(T)$ where maxg(T)/ming(T) is the maximum/minimum global time ticks within composite time stamp T. We want to show that if $maxming(e1) < maxming(e2) - 9$ and $T(e2) \sim T(e3)$ then we can guarantee that $T(e1) < T(e3)$ which means e1 is no long useful which can be flushed out. By proposition 5.1 we have $maxg(T(e2)) - ming(T(e3)) \leq 1g_g$ for any two composite time stamps concurrent to each other. So, if $T(e2) \sim T(e3)$, we have $\|maxming(T(e2)) - maxming(T(e3))\| \leq \|maxg(T(e2)) - maxg(T(e3))\| + \|ming(T(e2)) - ming(T(e3))\| \leq 1g_g + 1g_g = 2g_g$.

Since $maxming(T(e1)) < maxming(T(e2) - 9$,

we have $maxming(T(e1)) < maxming(T(e3) - 6$ which means $T(e1) < T(e3)$. Otherwise, if $T(e1) \sim T(e3)$ implies $maxming(T(e1)) - maxming(T(e3)) < 3$ and if $T(e1) \bowtie T(e3)$ implies $maxming(T(e1)) - maxming(T(e3)) < 5$

Notice that each event has to wait until all relevant events with smaller time stamps have arrived and each new arrival event may trigger one or more occurrence of the composite events (irrespective of the event is initiator or terminator). The context constraint can be enforced because when the event is evaluated, all events that participate in the evaluation have already arrived. On the other hand, due to concurrency of the events and delay of the propagation, the algorithm can become very complicated and the number of events that need to be buffered may be large. The algorithms for different operators with Recent context can be found in the appendix.

From the previous section we can see that the definition of the contexts gives us a general picture about how the events are collected and consumed. In the asynchronous evaluation policy, because the events are evaluated based only on the partial

knowledge of the current status, and because of the partial ordered composite time stamps, the concept of "most recent" and "oldest" events need to be redefined. For one thing, those "most recent" and "oldest" may not be unique any more because of the existence of concurrent events. Furthermore, the current "most recent" and "oldest" may not be valid if some events that have occurred earlier are delayed from different sites. One the other hand, event consumption becomes much more complicated. Using the asynchronous policy, the event consumption is based on the current status of the event occurrence (hence "unwanted" events at current stage may trigger a valid composite event in the future).

But the definition of context itself do not tell us exactly what to do when each event reaches the distributed event detector (or GED). For example, in recent context, there is no strict rule dealing with the arrival of an event (initiator or terminator). So, at least two policies can be considered.

1. When an initiator comes, it is just buffered and wait until the terminator to do the evaluation, propagation as well as flushing of the unwanted events.

2. When an initiator comes, the list of the initiator is alway maintained to have the most recent property (flush out the out-of-dated initiator). When the event terminator comes, it evaluate the event based on the event operator, and propagate the triggered composite event to the parent node.

It is easy to see that given a sequence of the events, the second one will detect more composite events than the first one. That is if an event is detected in the policy, it must be detected in the second one, but the reverse may not hold. Which one is "better" totally depends on the application. As an example, based on the above two policies, we have the following two different algorithms for SEQ operator:

1. SEQ(E1, E2):

if e1 signaled

insert e1 to E1's list;

E1's list = Max (E1's list);

if e2 signaled

if (E1's list != empty)

for all e1 in E1's list , propagate < e1, e2 > to parents node.

2. SEQ(E1, E2):

if e1 signaled

insert e1;

if e2 signaled

let SEQSet = e1 in E1 such that e1 < e2

if (SeqSet != empty)

let maxSeq = Max(SEQSet);

forall e1 in maxSeq, propagate < e1, e2 > to parents node.

delete all e1 in SeqSet;

Depending on the application, different algorithms can be derived based on the evaluation policies and context constraint.

6.4 Implementation

Based on the evaluation policies discussed last section, we can have several different algorithms and implementation. Due to the complexity and delayed evaluation properties of Synchronous evaluation, we choose the Asynchronous evaluation for implementation.

6.4.1 Algorithm

The algorithm is based on the Asynchronous evaluation policy from last section. As in last section, two different consideration can be associated with asynchronous evaluation policy. One is that when the event initiator comes, it is just buffered. The

event evaluation and consuming are done when the event terminator occurs. We call this "partial asynchronous" event evaluation policy. The "partial" here indicates the initiator has to "wait" compared with the other one. This one is a simple rule and easy to understand.

The other alternative is that when the event initiator comes, it starts to flush out unwanted events (if any); When the event terminator occurs, it starts the evaluation as well as consuming the unwanted events. The one is called "total asynchronous", again, for the comparison of the above one. One good thing about this policy is that it buffers least events and is very close to the local event detection algorithm.

Notice that the only difference between the above two is that the "total" one starts doing the event consumption when the initiator comes while the "partial" one waits until the terminator. Based on the context we have so far, only the Recent context may result different algorithm since only the "most recent" initiators need to be kept. In other contexts, such as Chronicle, Continuous and Cumulative, all initiators are kept for future (terminator) evaluation.

The following are the detailed description and the algorithms are in the appendix.

- **Recent-partial:** In this context, only the most recent occurrence of the initiator for any event that has started the detection of that event is used. When the initiator event occurs, the event is buffered; when the event terminator arrives, it finds the most recent set of concurrent events of the corresponding initiator and other constituent events satisfies the composite operator, and propagates to the parent nodes. At the same time, all the occurrences of events that cannot be the initiators of that event in the future are deleted (or flushed). In this context, *not all occurrences* of a constituent event will be used in detecting a composite event. Furthermore, an initiator of an event (primitive or composite) will continue to initiate new event occurrences until a new initiator occurs.

- **Recent-total:** In this context, only the most recent occurrence of the initiator for any event that has started the detection of that event is used. When an initiator arrives, the list of the initiator is alway maintained to have the most recent property (flush out the out-of-date initiator). Notice again there could be multiple "the most recent" initiators. When the event terminator arrives, it paired with the initiators (already the "most recent" at that moment) and other constituent events that satisfies the composite operator, and propagated to the parent nodes. At the same time, all occurrences of events that cannot be the initiators of that event in the future are deleted (or flushed). In this context, *not all occurrences* of a constituent event will be used in detecting a composite event. Furthermore, an initiator of an event (primitive or composite) will continue to initiate new event occurrences until a new initiator occurs.

- **Chronicle:** In this context, for an event occurrence, the initiator, terminator pair is unique. When an initiator comes, it is simply buffered. When the event terminator comes, it is paired with the oldest concurrent initiators along with the other constituent events that satisfy the operator and propagate to the parents node. In the meanwhile, the oldest concurrent initiator and those constituent events being propagated are flushed out.

- **Continuous:** In this context, each initiator of an event starts the detection of that event. When the initiator occurs, it is simply buffered. A terminator event occurrence may detect one or more occurrences of the same event. The initiator and the terminator are discarded after an event is detected. When the terminator occurs, it is paired with all initiator and other constituent and propagate to parents node. In the meanwhile, the initiators and all the constituent events being propagated are flushed out. This context is especially useful for tracking trends of interest on a sliding time point governed by the initiator event.

- **Cumulative:** In this context, all occurrences of an event type are accumulated as instances of that event until the event is detected. When the initiator occurs, it is simply buffered. Whenever the event terminator occurs and an event is detected, all the initiator and all constituent events that satisfy the event operator are cumulated and propagate to the parents node. In the meanwhile all the occurrences that are used for detecting that event are deleted.

The main difference between the centralized algorithm and the asynchronous ones is that There could be multiple concurrent initiators paired with the event terminator and propagate to the parents.

The algorithms can be simplified by

1. Keeping one out of possibly multiple concurrent time stamps to represent the composite time stamp of the event. Notice that keeping one time stamp will not effect the pseudo-algorithm.

2. Propagating just one constituent event out of possibly multiple concurrent ones to the parent node. This one will affect those context which "most recent" or "oldest" property need to hold when propagating the events. That means only recent and chronicle context will be different but continuous and cumulative context will remain same in the algorithm.

The algorithms (only the recent and chronicle) are also in the appendix.

6.4.2 Data Structure

The detailed Global Event Detection architecture is in [19]. Here the main modification is discussed.

PrimTS and compTS data structures are added to hold the primitive time stamps and composite time stamps. PrimTS structure consists of site, global time and local time. CompTS structure is a linked list of the concurrent maximum of the constituent

primTSs. These two structures are added into the $L\_OF\_L\_LIST$ and are propagated to the parents node when an event occurs. CompTS is used to compare and evaluate the composite events.

The primTS and compTS is initialized when the primitive event occurs. The site attribute is set to be the machine name and the local time is set to be the current system time. The global time is calculated by truncating the local time with desired precision.

When the composite event is evaluated, the compTS of the constituent event is the time stamp to be compared. Based on the algorithm, if the composite event occurs, the parameter list $L\_OF\_L\_LIST$ of the constituent events are merged and compTS are recomputed from the compTS of the constituent events to have the maximum and concurrent property.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

This thesis presents a mathematical framework of ordered sets relation for distributed composite event detection. A new definition of distributed composite time stamps is defined ensuring the "maximum" and "concurrency" properties of the constituent primitive time stamps. The strict ordering relation on the time stamps for comparison in a distributed system is chosen carefully to have the "least restrictive" property and proved it to be well-defined. A "Max" operator which is well-integrated with the definition of time stamps is introduced for handling and propagating the time stamps. A number of properties are rigorously proved for a better understanding of the semantics of distributed time stamps and the partial ordering. Finally, using the formalism developed, the semantics of Sentinel composite event operators is extended to the distributed environment. The main contributions of this thesis are:

- Introduce the *least restrictive strict partial ordering* of the set of the composite event time stamps.
- Introduce the *max* operator to describe the composite time stamps.
- Design algorithms for detecting composite events using different evaluation policies and parameter contexts.
- Implement the composite event detection algorithm in GED.

The future work can be the following:

- Re-examination of the event operators and context constraints for the real-world applications. Identification of the composite event operators and context constraints are needed for the real world applications.

- Generalize the context constraints to arbitrary functions that can be used as filters.

- The relationship between the event evaluation policies and context constraints. Applicability of evaluation policies to contexts. Some evaluation policies may be meaningful in some contexts.

- Containing event logs in distributed environments to detect complex global events (correlation of events).

APPENDIX
COMPOSITE EVENT DETECTION ALGORITHMS

In this appendix, we present the algorithms for the distributed event operators construction and detection.

Algorithm: Synchronous Policy, Recent Context:

OR:

```
if e1 signaled
  if E2 ! = empty
    delete and propagate all e2 such that e2 <  e1 to parents
    if exists e2 in E2 such that e1 < e2
      propagate e1 to parents
    else insert e1
  else insert e1
if e2 signaled
  Same as e1
```

AND:

```
if e1 signaled
  if E2 != empty
    if exists e2 in E2 such that e1 < e2
    //e2 up to dated , e1 can be evaluated with respect to e2
      let RE2 =  e2 in E2, e2<  e1 and
      //RE2 is the most recent set with respect to e1
      if RE2 != empty
        for all e2 in RE2, propagate < e1, e2 > to parents.
        insert e1
      delete all e2 with maxming(e2)<maxming(e1)-9
if e2 signaled
  Same as e1
```

SEQ:

```
if e1 signaled
  if E2 != empty and ( exists e2 in E2, e1 < e2 )
  //all relevant related e1 has arrived
  //all e20 with e20<e1 can be evaluated now.
    let SeqE2 =  e2 in E2 such that e2 < e1
    if SeqE2!= empty
      for all e2 in SeqE2
```

```
            let RE1 =  e1 in E1 such that e1 < e2
            //RE1 is the set of most recent e1<e2
                for all e1 in RE1
                propagate < e1, e2 > to parents
                delete e2 in SeqSet form E2
    insert e1

    if e2 signaled
      if E1 != empty and if exists e1 in E1 such that e2 < e1
      //all relevant e1 has arrived, e2 can be evaluated now
          let RE1 =  e1 in E1 such that e1 < e2
      //RE1 is the set of most recent e1< e2.
      for all e1 in RE1
        propagate < e1, e2 > to parents
      else insert e2
      delete all e1 with maxming(e1)<maxming(e2)-9

A, P:
    if e1 signaled
      if E2!=empty and if exists e2 in E2 such that maxming(e1)<maxming(e2)-9
      //this wired condition is to ensure all related e1
      //has arrived. this is due to E1<  E3 the unpleasant  .
        let AE = e2 in E2 such that maxming(e2)<maxming(e1)-9
        for all e2 in AE
          let RE2 =  e1 in E1 such that e1<  e2
          for all e1 in RE2
              propagate < e1,e2> to parents
          delete e2 from E2
    else insert e1

    if e2 signaled
      if e1 != empty and if exists e1 in E1 such that maxming(e2)<maxming(e1)-9
      //all relevant e1 has arrived. e2 can be evaluated now
        let RE = e1 in E1 such that e1<  e2 and e1 is the
          for all e1 in RE, propagate < e1, e2> to parents
      else insert e2

    if e3 signaled
      delete all e1 with maxming(e1)<maxming(e3)-9
      delete all e2 with maxming(e2)<maxming(e3)-9

A*, P*
    if e1 signaled
```

```
    if E3!=empty and if exists e3 in E3 such that e3 < e1
    //all e30 with e30<e1 can be considered to evaluate now.
      let Astar=e3 in E3 such that e3 < e1
      if Astar != empty
        for all e3 in Aster
            if exists e2 in E2, e3<e2
            //for this e3, e2 has already arrived
            //e3 now ready for full evaluation
              let RE1=e1 in E1, e1 is the most recent(maximum)one<e3
              for all e1 in RE1, propagate
                  <e1,all e2 such that e1<  e2<e3 ,e3> to parents
                  delete e3
  insert e1;

  if e2 signaled insert e2;

  if e3 signaled
    if E1 != empty and if exists e1 in E1 such that e3 < e1
    if E2 != empty and if exists e2 in E2 such that e3 < e2
    //all related e1 and e2 has arrived
    //now e3 is ready to be evaluated
      let RE1 =e1 in E1, e1 is the most recent(maximum)one<e3
      for all e1 in RE1, propagate
        <e1,all e2 such that e1<  e2<e3 ,e3> to parents
    else insert e3.
  delete all e1 with maxming(e1)<maxming(e3)-9
  delete all e2 with maxming(e2)<maxming(e3)-9

NOT:
  if e1 signaled
    if E3!=empty and if exists e3 in E3 such that e3 < e1
    //all e30 with e30<e1 can be considered to evaluate now
      let Astar=e3 in E3 such that e3 < e1
      if Astar != empty
        for all e3 in Aster
        if exists e2 in E2, e3<e2
        //for this e3, e2 has already arrived. e3 now ready for evaluation
            let RE1=e1 in E1, e1 is the most recent(maximum)one<e3
            for all e1 in RE1,
                if no exists e2 such that e1<  e2<e3
                  propagate <e1,e3> to parents
                  delete e3
    insert e1;
```

```
if e2 signaled insert e2;
if e3 signaled
  if E1 != empty and if exists e1 in E1 such that e3 < e1 and
  if E2 != empty and if exists e2 in E2 such that e3 < e2 and
  //all related e1 and e2 has arrived
  //now e3 is ready to be evaluated
    let RE1 =e1 in E1, e1 is the most recent(maximum)one<e3
    for all e1 in RE1,if no exists e2 such that e1<  e2<e3
      propagate <e1,e3> to parents
    else insert e3.
  delete all e1 with maxming(e1)<maxming(e3)-9
  delete all e2 with maxming(e2)<maxming(e3)-9
```

Algorithm: "Partial" Asynchronous Policy, Recent Context

```
AND(E1, E2):
    if left event e1 is signaled
      if E2's list is not empty,
        let ANDE2=e2 in E2, e2< e1
        let maxANDE2=Max(ANDE2);
        forall e2 in maxANDE2
            propagate < e1,e2> to parents node;
            delete all e2 such that e2<e1;
    insert e1;

    if e2 signaled
      if E1 != empty,
        let ANDE1=e1 in E1, e1<  e2
        let maxANDE1=Max(ANDSet);
        for all e1 in maxANDE1 , propagate < e1,e2> to parents node;
            delete all e1, e1<e2;
    insert e2;

OR(E1, E2):
    if e1 signaled , propagate < e1 > to parents node.
    if e2 signaled , propagate < e1 > to parents node.

SEQ(E1, E2):
    if e1 signaled
      insert e1;

    if e2 signaled
      let SEQSet =  e1 in E1 such that e1 < e2
      if ( SeqSet != empty )
        let maxSeq = Max(SEQSet);
        for all e1 in maxSeq, propagate < e1, e2 > to parents node.
        delete all e1 in SeqSet and insert maxSeq (keep the initiator);

A, P(E1, E2, E3):
    if e1 signaled
      insert e1

    if e2 signaled
      let ASet =  e1 in E1 such that e1 <  e2
      let maxASet = Max(ASet);
      if maxAset != empty
        for all e1 in maxASet propagate < e1, e2 > to parents node
```

```
            delete all e1 in ASet and insert maxASet (initator);

      if e3 signaled
        delete all e1 in E1 such that e1 < e3;

A*, P*(E1, E2, E3):
      if e1 signaled
        insert e1;

      if e2 signaled,
        insert e2;

      if e3 signaled
        let ASet =  e1 in E1 such that e1 < e2
        let maxASet = Max(ASet);
        if maxASet != empty
          for all e1 in maxASet,
            let E2Set = e2 in E2, e1 <   e2 < e3 ;
            if E2Set != empty
              propagate < e1, all e2 in E2Set, e3 > to parents node
              delete all e1 < e3, all e2 < e3

NOT(E1, E2, E3):
      if e1 signaled
        insert e1;

      if e2 signaled
        insert e2 in E2;

      if e3 signaled
        let NOTSet =  e1 in E1 such that e1 < e3
        let maxNOTSet = Max(NOTSet);
        if MaxNOTSet != empty
          let e1 in MaxNOTSet;
          if not exists e2 in E2 such that e1 < e2 <e3
            then propagate < e1, e3 > to parents.
            delete all e1 < e3 and all e2<e3
```

Algorithm: "Total" Asynchronous Policy, Recent Context

```
AND(E1, E2):
    if left event e1 is signaled
      if E2's list is not empty,
        forall e2 in E2 , propagate < e1,e2> to parents node;
        insert e1 to E1's list;
      E1's list = Max ( E1's list );

    if e2 signaled
      if E1's list is not empty,
        for all e1 in E1 , propagate < e1,e2> to parents node;
        insert e2 to E2's list;
      E2's list = Max ( E2's list );

OR(E1, E2):
    if e1 signaled , propagate < e1> to parents node.
    if e2 signaled , propagate < e1> to parents node.

SEQ(E1, E2):
    if e1 signaled
      insert e1 to E1's list;
      E1's list = Max ( E1's list );

    if e2 signaled
      if ( E1's list != empty )
      for all e1 in E1's list such that e1 < e2,
        propagate < e1, e2 > to parents node.

A, P(E1, E2, E3):
    if e1 signaled
      insert e1 to E1's list;
      E1's list = Max ( E1's list );

    if e2 signaled
      if E1's list is not empty
        for all e1 in E1's list such that e1 < e2,
            propagate < e1, e2 > to parents node

    if e3 signaled
      delete all e1 in E1 such that e1 < e3;

A*, P*(E1, E2, E3):
    if e1 signaled
```

```
        insert e1 to E1's list;
        E1's list = Max ( E1's list );
        delete all e2 with e1 < e2;

      if e2 signaled,
        if E1 is not empty and if for all e1 we have e1 < e2
          insert e2;

      if e3 signaled
        if E2Less =  e2 in E2 such that e2 < e3 is not empty
          for all e1 in E1, propagate < e1, all e2 in E2Less, e3 > to parents node.
              delete all e1 in E1, e2 in E2 such that e1 < e3 or e2 < e3.
        else
          for all e1 in E1, propagate < e1, e3 > to parents node.
              delete all e1 in E1 such that e1 < e3.

NOT(E1, E2, E3):
      if e1 signaled
        insert e1 to E1's list;
        E1's list = Max ( E1's list );

      if e2 signaled
        if E1 is not empty
          delete all e1 in E1 such that e1 < e2

      if e3 signaled
        if E1 is not empty
          for all e1 in E1 such that e1 < e3, propagate < e1, e3 > to parents.
        delete all e1 < e3.
```

Algorithms: Asynchronous Policy, Chronicle Context

```
OR(E1, E2):
    if e1 signaled , propagate < e1 > to parents node.
    if e2 signaled , propagate < e1 > to parents node.

AND(E1, E2):
    if e1 signaled
      if E2 != empty,
        forevery e2 in E2 , propagate < e1,e2> to parents node;
          delete e2 in E2;
      else insert e1;

    if e2 signaled
      if E1 != empty,
        for every e1 in E1 , propagate < e1,e2> to parents node;
          delete e1 in E1;
      else insert e2;

SEQ(E1, E2):
    if e1 signaled
      insert e1;
    if e2 signaled
      let SEQSet =  e1 in E1 such that e1 < e2
      if ( SeqSet != empty )
        let minSeq = min(SEQSet);
        for every e1 in minSeq, propagate < e1, e2 > to parents node.
          delete all e1 in minSeq;

A, P(E1, E2, E3):
    if e1 signaled
      insert e1

    if e2 signaled
      let ASet =  e1 in E1 such that e1 <  e2
      let minASet = Min(ASet);
      if minAset != empty
        for every e1 in minASet propagate < e1, e2 > to parents node
          delete all e1 in minASet;

    if e3 signaled
      delete all e1 in E1 such that e1 < e3;

A*, P*(E1, E2, E3):
```

```
    if e1 signaled
      insert e1;

    if e2 signaled,
      insert e2;

    if e3 signaled
      let ASet =  e1 in E1 such that e1 < e2
      let minASet = Min(ASet);
      if minASet != empty
        for every e1 in minASet,
          let E2Set = e2 in E2, e1 <   e2 < e3 ;
          if E2Set != empty
            propagate < e1, all e2 in E2Set, e3 > to parents node
        delete all e1 < e3 and all e2 < e3

NOT(E1, E2, E3):
    if e1 signaled
      insert e1;

    if e2 signaled
      if E1 is not empty
        delete all e1 such that e1 < e2;

    if e3 signaled
      let NOTSet =  e1 in E1 such that e1 < e3
      let minNOTSet = Min(NOTSet);
      if minNOTSet != empty
        let e1 in minNOTSet;
        propagate < e1, e3 > to parents.
      delete all e1 < e3
```

Algorithm for the Asynchronous Policy, Continuous Context

```
OR(E1, E2):
    if e1 signaled , propagate < e1 > to parents node.
    if e2 signaled , propagate < e1> to parents node.

AND(E1, E2):
    if e1 signaled
      if E2 != empty,
        for every e2 in E2 , propagate <e2,e1> to parents node;
        flush E2's buffer;
      else insert e1;

    if e2 signaled
      if E1 != empty,
        for every e1 in E1 , propagate < e1,e2> to parents node;
        flush E1's buffer;
      else insert e2;

SEQ(E1, E2):
    if e1 signaled
      insert e1;
    if e2 signaled
      let SEQSet =  e1 in E1 such that e1 < e2
      if ( SeqSet != empty )
        for every e1 in SeqSet, propagate < e1, e2 > to parents node.
            delete all e1 in SeqSet;

A, P(E1, E2, E3):
    if e1 signaled
      insert e1

    if e2 signaled
      let ASet =  e1 in E1 such that e1 <  e2
      if Aset != empty
        for every e1 in ASet propagate < e1, e2 > to parents node

    if e3 signaled
      delete all e1 in E1 such that e1 < e3;

A*, P*(E1, E2, E3):
    if e1 signaled
      insert e1;
```

```
    if e2 signaled,
      if E1 is not empty and E1's head < e2
        insert e2;

    if e3 signaled
      let ASet =  e1 in E1 such that e1 < e3
      if ASet != empty
        for every e1 in ASet,
        let E2Set = e2 in E2, e1 <   e2 < e3 ;
        if E2Set != empty
            propagate < e1, all e2 in E2Set, e3 > to parents node
      else
        propagate < e1, e3 > to parents node
      delete all e1 < e3 and all e2 < e3

NOT(E1, E2, E3):
    if e1 signaled
      insert e1;

    if e2 signaled
      if E1 is not empty
        delete e1 such that e1 < e2;

    if e3 signaled
      let NOTSet =  e1 in E1 such that e1 < e3
      if NOTSet != empty
        for each e1 in NOTSet;
            propagate < e1, e3 > to parents.
      delete all e1 < e3
```

Algorithm: Asynchronous Policy, Cumulative Context

```
OR(E1, E2):
    if e1 signaled , propagate < e1 > to parents node.
    if e2 signaled , propagate < e1 > to parents node.

AND(E1, E2):
    if e1 signaled
      if E2 != empty,
        propagate < all e2 in E2, e1> to parents node;
        flush E2's buffer;
      else insert e1;

    if e2 signaled
      if E1 != empty,
        propagate < all e1 in E1,e2> to parents node;
        flush E1's buffer;
      else insert e2;

SEQ(E1, E2):
    if e1 signaled
      insert e1;
    if e2 signaled
      let SEQSet =  e1 in E1 such that e1 < e2
      if ( SeqSet != empty )
        propagate < all e1 in SEQSet, e2 > to parents node.
        delete all e1 in SeqSet;

A, P(E1, E2, E3):
    if e1 signaled
      insert e1

    if e2 signaled
      let ASet =  e1 in E1 such that e1 <  e2
      if Aset != empty
        propagate < all e1 in ASet, e2 > to parents node
        delete e1 in ASet;

    if e3 signaled
      delete all e1 in E1 such that e1 < e3;

A*, P*(E1, E2, E3):
    if e1 signaled
      insert e1;
```

```
    if e2 signaled,
      if E1 is not empty and E1's head<e2
        insert e2;

    if e3 signaled
      let ASet =  e1 in E1 such that e1 < e2
      if ASet != empty
        let E2Set = e2 in E2, e1<  e2<e3 for all e1 in Aset
        if E2Set != empty
          propagate < all e1 in Aset, all e2 in E2Set, e3 ¿ to parents node
          delete all e1 < e3 and all e2 < e3
      else
        propagate < all e1 in Aset, e3 > to parents node
        delete all e1 < e3

NOT(E1, E2, E3):
    if e1 signaled
      insert e1;

    if e2 signaled
      if E1 is not empty
        delete all e1 such that e1 < e2;

    if e3 signaled
      let NOTSet =  e1 in E1 such that e1 < e3
      if NOTSet != empty
        propagate < all e1 in NOTSet, e3 > to parents.
        delete all e1 < e3
```

Algorithm: Simplified "Total" Asynchronous Policy, Recent Context

procedure Max1(Elist, e) : find the max one time stamp in Elist and e.

```
AND(E1, E2):
    if left event e1 is signaled
      if E2's list is not empty,
        propagate < e1,e2> to parents node;
      E1's list = Max1 ( E1's list, e1 );

    if e2 signaled
      if E1's list is not empty,
        propagate < e1,e2> to parents node;
      E2's list = Max1 ( E2's list, e2 );

OR(E1, E2):
    if e1 signaled , propagate < e1 > to parents node.
    if e2 signaled , propagate < e1 > to parents node.

SEQ(E1, E2):
    if e1 signaled
      E1's list = Max1 ( E1's list, e1 );

    if e2 signaled
      if ( E1's list != empty ) and (e1 = E1's head < e2 )
        propagate < e1, e2 > to parents node.

A, P(E1, E2, E3):
    if e1 signaled
      E1's list = Max1 ( E1's list, e1 );

    if e2 signaled
      if E1's list is not empty and e1 = E1's head < e2
        propagate < e1, e2 > to parents node

    if e3 signaled
      delete all e1 in E1 such that e1 < e3;

A*, P*(E1, E2, E3):
    if e1 signaled
      E1's list = Max1 ( E1's list, e1 );
      delete all e2 with e1 < e2;

    if e2 signaled,
```

```
        if E1 is not empty and if E1's head < e2
        insert e2;

    if e3 signaled and e1= E1's head < e3
        if E2Less =  e2 in E2 such that e2 < e3 is not empty
            propagate < E1's head, all e2 in E2Less, e3 > to parents node;
            flush E1, delete e2 in E2 such that e2 < e1;
        else
            propagate < E1's head, e3 > to parents node;
        flush E1;

NOT(E1, E2, E3):
    if e1 signaled
        E1's list = Max1 ( E1's list, e1 );

    if e2 signaled
        if E1 is not empty
            flush E1 if E1's head < e2

    if e3 signaled
        if E1 is not empty and E1's head < e3
            propagate < E1's head, e3 > to parents
        flush E1
```

Algorithm: Simplified "Total" Asynchronous Policy, Chronicle Context

min1(Elist): return the one of the event from Elist with the minimum time stamp.
minLess1(Elist, e): return the one of the event from Elist with the minimum time stamp that less than e.

```
OR(E1, E2):
    if e1 signaled , propagate < e1 > to parents node.
    if e2 signaled , propagate < e1 > to parents node.

AND(E1, E2):
    if e1 signaled
      ifE2 != empty,
       let e2 = min1(E2);
       if ( e2 != NULL )
          propagate < e1,e2> to parents node;
       delete e2 in E2;
      else insert e1;

    if e2 signaled
      if E1 != empty,
       let e1 = min1(E1);
       if ( e1 != NULL )
          propagate < e1,e2> to parents node;
       delete e1 in E1;
      else insert e2;

SEQ(E1, E2):
    if e1 signaled
      insert e1;
    if e2 signaled
      let e1 = minLess1(E1, e2)
      if ( e1 != NULL )
       propagate < e1, e2 > to parents node.
       delete e1 in E1;

A, P(E1, E2, E3):
    if e1 signaled
      insert e1

    if e2 signaled
      let e1 = minLess1(E1, e2)
      if e1 != NULL
       propagate < e1, e2 > to parents node;
      delete e1 in E1;
```

```
    if e3 signaled
      delete all e1 in E1 such that e1 < e3;

A*, P*(E1, E2, E3):
    if e1 signaled
      insert e1;

    if e2 signaled,
      if E1 is not empty and E1's head < e2
       insert e2;

    if e3 signaled
      let e1 = minLess1(E1, e3);
      if e1 != NULL
       let E2Set = e2 in E2, e1 <   e2 < e3 ;
       if E2Set != empty
          propagate < e1, all e2 in E2Set, e3 > to parents node
       else propagate < e1, e3 > to parents node
      delete e1, delete all e2 < e3

NOT(E1, E2, E3):
    if e1 signaled
      insert e1;
    if e2 signaled
      if E1 is not empty
       delete all e1 such that e1 < e2;
    if e3 signaled
      let e1 = minLess1(E1, e3);
      if e1 != NULL
       propagate < e1, e3 > to parents.
      delete e1;
```

REFERENCES

[1] U. Dayal, B. Blaustein and A.P. Buchmann, "The HiPAC project: Combining active databases and timing constraints," SIGMOD RECORD, 17(1), March 1988.

[2] N. Gehani and H. V. Jagadish, "Ode as an Active Database:Constraints and Triggers," Proceedings of the International Conference on VLDB, pp.327-336, September, 1991.

[3] N. H. Gehani, H. V. Jagadish and O. Shmueli, "COMPOSE: A System For Composite Event Specification and Detection," Technical report, AT&T Bell Laboratories, Murray Hill, NJ, December 1992.

[4] O. Diaz, N. Paton and P. Gray, "Rule Management in Object-Oriented Databases: A Unified Approach," In Proceedings 17th International Conference on Very Large Data Bases, Barcelona (Catalonia), Spain, Sept. 1991.

[5] S. Gatziu and K.R. Dittrich, "Samos: An active object-oriented database system," IEEE Quarterly Bulletin on Data Engineering, March 1993.

[6] S. Gatziu and K. R. Dittrich, "Events in an Active Object-Oriented Database System," Proceedings of 1st International Workshop on Rules in Database Systems, Edinburgh, August, 1993.

[7] S. Gatziu and Dittrich K. R., "Detecting Composite Events in Active Databases using Petri Nets," In Proc. of the 4th International Workshop on Researching Issues in Data Engineering: Active Database Systems(RIDE-ADS),pp. 2-9, February, 1994.

[8] D. Mishra, SNOOP: An Event Specification Language for Active Databases, Master's thesis, University of Florida, Gainesville, 1991.

[9] S. Charkravathy, V. Krishnaprasad, E Anwar and S.-k. Kim, "Composite Events for Active Databases: Semantics, Contexts and Detection," Proceedings of the 20th VLDB Conference, pp. 606-617, Santiago, Chile, 1994.

[10] S. Chakravarthy and D. Mishra, "Snoop: An Expressive Event Specification Language for Active Databases," Data and Knowledge Engineering, 14(10):1–26, October 1994.

[11] S. Schwiderski, Monitoring the Behaviour of Distributed Systems, Ph.D thesis, University of Cambridge, London, 1996.

[12] R. Orfali, D. Harkey and J. Edward, The Essential Distributed Objects Survival Guide, John Wiley & Sons, Inc., NJ, 1996.

[13] Object Management Group, Framingham, MA, CORBAServices: Common Object Services Specification v1.0. March 1995.

[14] Anton V. Schedl, SNOOP: Design and Simulation of Clock Synchronization in Distributed Systems, Master's thesis, Technische Universität Wien, Institue für Technische Informatik, 1996.

[15] S. Chakravarthy and D. Mishra, "Towards an Expressive Event Specification Language for Active Databases," In Proceedings of the 5th International Hong Kong Computer Society Database Workshop on Next generation Database Systems, Kowloon Shangri-La, Hong Kong, February 1994.

[16] H. Kopetz," Sparse time versus dense time in distributed real-time systems," Proceedings of the 12th International Conference on Distributed Computing Systems, Yokohama, Japan, June 1992.

[17] H. Kopetz and K.H. Kim," Real-time objects and temporal uncertainties in distributed computer systems," Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems, Huntsville, AL, 1990.

[18] V. Krishnaprasad, Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation, Master's thesis, University of Florida, Gainesville, 1994.

[19] H. Liao, Global Event Detection in Sentinel: Design and Implementation of A Global Event Detector, Master's thesis, University of Florida, Gainesville, 1997.

BIOGRAPHICAL SKETCH

Shuang Yang was born on June 29, 1968, at Fuzhou, China. She received her undergraduate degree in mathematics from Beijing Normal University, China, in July 1990. She also received her master's degree in multivariate statistics from Fuzhou University, China, in June 1993 and master's degree in applied mathematics from University of Florida. She will receive her Master of Science degree in computer and information sciences from the University of Florida, Gainesville, in May 1999. Her research interests include active, object-oriented and heterogeneous databases.