Technical Report

UCAM-CL-TR-400
ISSN 1476-2986

Number 400

Monitoring the behaviour of distributed systems

Scarlet Schwiderski

July 1996

This technical report is based on a dissertation submitted April 1996 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Selwyn College.

ISSN 1476-2986

Abstract

Monitoring the behaviour of computing systems is an important task. In active database systems, a detected system behaviour leads to the triggering of an ECA (event-condition-action) rule. ECA rules are employed for supporting database management system functions as well as external applications. Although distributed database systems are becoming more commonplace, active database research has to date focussed on centralised systems. In distributed debugging systems, a detected system behaviour is compared with the expected system behaviour. Differences illustrate erroneous behaviour. In both application areas, system behaviours are specified in terms of events: primitive events represent elementary occurrences and composite events represent complex occurrence patterns. At system runtime, specified primitive and composite events are monitored and event occurrences are detected. However, in active database systems events are monitored in terms of physical time and in distributed debugging systems events are monitored in terms of logical time. The notion of physical time is difficult in distributed systems because of their special characteristics: no global time, network delays, etc.

This dissertation is concerned with monitoring the behaviour of distributed systems in terms of physical time, i.e. the syntax, the semantics, the detection, and the implementation of events are considered.

The syntax of primitive and composite events is derived from the work of both active database systems and distributed debugging systems; differences and necessities are highlighted.

The semantics of primitive and composite events establishes when and where an event occurs; the semantics depends largely on the notion of physical time in distributed systems. Based on the model for an approximated global time base, the ordering of events in distributed systems is considered, and the structure and handling of timestamps are illustrated. In specific applications, a simplified version of the semantics can be applied which is easier and therefore more efficient to implement.

Algorithms for the detection of composite events at system runtime are developed; event detectors are distributed to arbitrary sites and composite events are evaluated concurrently. Two different evaluation policies are examined: asynchronous evaluation and synchronous evaluation. Asynchronous evaluation is characterised by the ad hoc consumption of signalled event occurrences. However, since the signalling of events involves variable delays, the events may not be evaluated in the system-wide order of their occurrence. On the other hand, synchronous evaluation enforces events to be evaluated in the system-wide order of their occurrence. But, due to site failures and network congestion, the evaluation may block on a fairly long-term basis.

The prototype implementation realises the algorithms for the detection of composite events with both asynchronous and synchronous evaluation. For the purpose of testing, primitive event occurrences are simulated by distributed event simulators. Several tests are performed illustrating the differences between asynchronous and synchronous evaluation: the first is ‘fast and unreliable’ whereas the latter is ‘slow and reliable’.

To my parents

Preface

The research described in this dissertation was carried out in the University of Cambridge Computer Laboratory under the supervision of Dr. Ken Moody, beginning in October 1992. I would like to acknowledge the support of many.

Above all, it is a pleasure to express my gratitude to Dr. Ken Moody, who guided me through the course of this study. It is due to his patient encouragement and persistence that I was nudged from theory towards practice. In October 1994, Ken started a sabbatical and passed my supervision to Dr. Andrew Herbert, the head of APM Ltd., Cambridge. I benefited greatly from Andrew's scientific knowledge and industrial experience on distributed object computing. I am fortunate that he continued to be my "assistant supervisor" up to the completion of this study.

At the Computer Laboratory, I was part of the Opera group, led by Dr. Jean Bacon and Dr. Ken Moody. I am very grateful to all members of the group, especially to Dr. Jean Bacon, Dr. Noha Adly, Mohamad Afshar, Dr. John Bates, Dr. Sai Lai Lo, Richard Hayton, Oliver Seidel, and Robert Sultana, for helpful discussions and practical advice.

I am obliged to Prof. Klaus Dittrich and Dr. Stella Gatziu from the database technology research group at Zürich University for their support in the past two years. They helped me to understand active database systems. Moreover, Klaus made it possible for me to present my work on several occasions and get the necessary feedback.

Without the help of Quentin Stafford-Fraser and Dr. David Evers it would not have been possible to master Modula-3 for Network Objects, which I used for the implementation of a prototype system. I would like to thank them for always answering my questions patiently.

I am grateful also to Dr. Martyn Johnson for his lessons on "time in distributed systems" and "communication protocols". It was very inspiring to learn about the systems-side of distributed computing.

My time at the Computer Laboratory would not have been so valuable without knowing Lewis Tiffany, the librarian. I am happy that he was always there when I needed help and advice.

This work was supported by grants from the German Academic Exchange Service and the European Community (Human Capital and Mobility).

Finally, I would like to thank Malte for being so wonderful.

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration. This dissertation is not substantially the same as any that I have submitted for a degree or diploma or other qualification at any other University. No part of this dissertation has already been, or is concurrently being, submitted for any such degree, diploma or other qualification. This dissertation does not exceed sixty thousand words, including tables, footnotes and bibliography.

Scarlet Schwiderski
Selwyn College, Cambridge
April 1996

Trademarks

Contents

List of Figures

List of Tables

Notation

General

E_n	event type	10
e_n	event occurrence of type E_n	10
e_n^m	m-th event occurrence of type E_n	10
$site \triangleright$	site of origin	50
$\perp$	undefined	73
$GT(E)$	global event tree of type E	92
Δ_{local}	delay of local event detection	104

Time and Order

Π	precision of synchronised local clocks	67
g_z	granularity of reference clock	67
g_g	granularity of global time base	67
$\rightarrow$	happened-before (causal order)	29
$\parallel$	concurrency (causal order)	29
$\rightarrow_{2g_g}$	$2g_g$-restricted temporal order	69
$\parallel_{2g_g}$	$2g_g$-restricted concurrency	69
$<$	happened-before relationship between timestamps	
	• general semantic model	76
	• simplified semantic model	82
$\sim$	concurrency relationship between timestamps	
	• general semantic model	76
	• simplified semantic model	82
$\bowtie$	unrelatedness between timestamps	77

Event Operators

,	conjunction	53
$\|$	disjunction	53
;	sequence	53
$|$$|$	concurrency	53
$\star$	iteration (≥ 0)	53
$+$	iteration (≥ 1)	53
NOT	negation	53

Timestamps

lt_k	local clock tick at site k	71
$gt_k(lt_k)$	global time of local clock tick lt_k	72
$T(e)$	timestamp of event e	
	• general semantic model	72
	• simplified semantic model	81
$T(e_1) \cup T(e_2)$	joined timestamp of events e_1 and e_2	
	• general semantic model	78
	• simplified semantic model	83
$base(T(E))$	base of timestamp $T(E)$	72
$limit(T(E))$	limit of timestamp $T(E)$	72
t	component timestamp	
	• general semantic model	73
	• simplified semantic model	82

Chapter 1

Introduction

1.1 Research Background

Most modern computing applications are distributed in nature. Such applications require support by distributed computing systems. Whilst each site in a distributed computing system may individually be simple, interaction between sites can create enormous complexity. Distributed system users, administrators, and developers require tools that enable them to monitor the behaviour of the distributed system as a whole. Event-driven monitoring offers a promising approach for observing how distributed computing systems behave. In this framework, the elementary observables of distributed system behaviour are identified as "primitive events": complex behaviour patterns of interest to users may be expressed in terms of them as "composite events". When the system is running, primitive events are detected and notified at registered event monitors, where they participate in the detection of composite events. It is thus possible to record the occurrence of complex behaviour patterns in which some agent has expressed an interest.

Event-driven monitoring covers a wide range of possible applications ranging from user-oriented applications to low-level system management. For example, in banking and stock exchange applications event-driven monitoring can be used to monitor the happenings on the money market in order to support financial trading and arbitrage. This both eases the workload on employees and provides assurance of integrity to authorities and institutions. Event-driven monitoring might also be used for integrity surveillance in safety-critical systems, such as automated industrial plants or gas- and oil-distribution networks.

1.2 Research Motivation

Present solutions to monitoring the behaviour of distributed systems are not adequate. Microsoft's COM(Component Object Model) provides a rudimentary event service called *connectable objects* [**?**, **?**]. The sources of events, the so-called connectable objects, notify event occurrences at registered consumers, the so-called *event sinks*. Events relate to changes to data, changes to the views on data, renaming of objects, clicking the mouse, or any action that changes control. There is no notion of composite events. In CORBA(Common Object Request Broker Architecture)'s event service, *suppliers* notify event occurrences at *consumers* via an *event channel* [**?**, **?**]. The event channel is a service that decouples the communication between suppliers and consumers. Hence, suppliers and consumers do not have to know about each other, but only about a single well-known event channel object. Two different event notification models are supported: the *push model* and the *pull model*. In the first, the supplier initiates the transfer of event data to consumers. In the latter, the consumer requests event data from suppliers. In [**?**, page 4-5] it is mentioned that "complex events may be handled by constructing a notification tree of event consumer/suppliers checking for successively more specific event predicates". However, this is not a general composite event service.

This dissertation presents a general solution to monitoring the behaviour of distributed systems. Primitive events can relate to physical time, to occurrences inside database systems or application programs, and to arbitrary external signals (e.g. from users or sensors). Composite events are complex patterns of primitive events, defined using an event algebra with well-defined semantics. Thus, composite events consist of primitive events and event operators, indicating the overall occurrence context. Specified primitive and composite events are monitored at arbitrary sites of a distributed system. Detected events cause reactions, for example, notifying users (as in the financial trading example) or taking some emergency measure (as in the safety-critical example).

Event-driven monitoring has the following advantages:

- *It is suitable for monitoring both internal and external system behaviour.* Employing primitive events relating to physical time in a distributed system with synchronised local clocks makes it possible to evaluate happenings within the system with respect to happenings within the system's environment.

- *The specification of system behaviour in terms of events is convenient.*

"An event represents the occurrence of significant system behaviour" [?, page 4]. Hence, it is convenient to specify system behaviour in terms of events. In state-driven approaches to monitoring, system behaviour can only be established by regularly polling the local states at the individual sites of a distributed system.

- *It is suitable for integrating existing (possibly heterogeneous) distributed systems with little change.* Each site is extended with an event interface, stating the event types originating at that site, and with measures for detecting occurrences regarding these event types. Event-monitoring modules can then be simply plugged in. In a heterogeneous distributed system, the application of a common interface definition language (IDL) is required for the specification of event interfaces.

- *Monitoring mechanisms are adaptable to system evolution.* In event-driven monitoring, a desired system behaviour is expressed declaratively rather than procedurally. In other words, what is described is the behaviour that is to be monitored, and not how this behaviour is to be monitored. Therefore, the system can decide on the best monitoring strategy and adapt in the case of system evolution.

1.3 Research Tasks

The research outlined in this dissertation is divided into the following tasks:

- *Syntax of events*
  The syntax of events determines how primitive and composite events are specified. Both primitive and composite events have parameters which capture the circumstances in which the event occurs. Depending on the kind of event, parameters are system- or user-defined. The kinds of primitive events and the event operators needed for specifying composite events are to be identified and a formal syntax for specifying events is to be developed. The formal parameters of events are to be investigated.

- *Semantics of events*
  A primitive or composite event expression describes a desired system behaviour. The semantics of events determines what exactly this behaviour is, i.e. when and where it occurs. For that reason, each occurrence of an event is

associated with a timestamp. Before defining the formal semantics of events, the notions of time and order in distributed systems are to be explored and the structure and handling of timestamps are to be investigated.

- *Detection of events*
  At system runtime, specified primitive and composite events are monitored and event occurrences are detected. There are different possibilities for dealing with the special characteristics of distributed systems. An architecture for a distributed event monitoring system is to be defined and generic algorithms for the detection of composite events are to be developed.

- *Prototype implementation*
  The practicability of the theoretical results is to be demonstrated and measured. A prototype is to be implemented which realises the detection algorithms defined in the formal model.

1.4 Research Context

Two research areas are of direct relevance to this work: active database systems and distributed debugging systems. Both research areas use primitive and composite events for monitoring relevant system behaviour. In active database systems, a detected system behaviour leads to the triggering of an ECA (event-condition-action) rule. ECA rules are employed for supporting database management system functions as well as external applications. Although distributed database systems are becoming commonplace, active database research has to date focussed on centralised systems. In distributed debugging systems, a detected system behaviour is compared with the expected system behaviour. Differences illustrate erroneous behaviour. Distributed debugging systems focus on monitoring the internal system behaviour only.

The field of distributed object computing provides a necessary infrastructure for representing and manipulating events in distributed systems (e.g. for communicating detected events and for garbage collecting obsolete events). This infrastructure is employed for the prototype implementation.

1.5 Dissertation Outline

Chapter 2 reviews the research area of active database systems. The general notion of an event is discussed and the most prominent research projects are examined. One common feature of all research projects is that they consider centralised systems only. The differences from distributed systems are identified and some related work is pointed out.

The research area of distributed debugging systems is reviewed in Chapter 3. A brief background on distributed systems is given, before the most prominent research projects are presented. Since approaches to distributed debugging vary considerably, a formal comparison is made between them.

The analysis of the related work and the goals of this dissertation are presented in Chapter 4.

The syntax of primitive and composite events is the subject of Chapter 5. A brief discussion of event parameters is included.

Chapter 6 establishes the semantics of primitive and composite events in distributed systems. The semantics depends on the notions of time and order in distributed systems. Two different semantic models are introduced: the general semantic model and the simplified semantic model. The latter is applicable in specific distributed systems, where the frequency of event occurrences is lower than some system-specific value. The structure and handling of timestamps concerning physical time are illustrated for both models, before the semantics is formally defined.

The detection of events is discussed in Chapters 7 and 8. Chapter 7 identifies the goals of event detection, defines the system architecture, and explains the basic detection mechanisms. The algorithms for the detection of composite events at system runtime are developed in Chapter 8. Two different evaluation policies are examined representing different approaches for dealing with the special characteristics of distributed systems: asynchronous evaluation and synchronous evaluation.

Chapter 9 describes a prototype implementation of the detection algorithms with both asynchronous and synchronous evaluation. Several tests are performed illustrating the differences between the two evaluation policies.

Chapter 10 concludes this dissertation with a summary and some suggestions for further work.

Chapter 2

Events in Active Database Systems

A database system is a system to model, store, and retrieve large volumes of data; multiple users manipulate the data at the same time. Conventional database systems are *passive*: data is created, updated, deleted, and retrieved only in response to operations issued outside the database system, either by users or by application programs. *Active* database systems enhance the functionality of conventional database systems: the database system itself issues operations in response to certain events occurring or certain conditions being satisfied. Active capability is captured by the *ECA (Event-Condition-Action) rule* paradigm: when an *event* is detected an *action* is triggered, if the *condition* holds. The event-part of an ECA rule specifies what causes the rule to be triggered. Once the rule is triggered, the condition-part is considered. A condition is a database predicate or a database query evaluating the database state at the time of event detection. If the database predicate is true or the database query produces a non-empty answer, the action in the action-part is executed. An action is an executable program, which may contain data modification or data retrieval operations, transaction operations, or arbitrary procedure calls. A short introduction into the field of active database systems is given in [**?**], whereas [**?**] contains a broad overview including a discussion of the application areas integrity constraint management, view maintenance, workflow management, and energy management.

The purpose of this chapter is to present the concepts of events in active database systems; that is, the chapter focusses on the event-part of ECA rules. The notion of an event is motivated in Section 2.1. The most prominent research projects are addressed thereafter. The research projects differ in their means to specify and detect events. The feasibility of active database research with respect to distributed computing environments is investigated in Section 2.7 and the particular deficiencies of current research efforts are highlighted.

2.1 The Notion of an Event

2.1.1 Primitive Events Versus Composite Events

An event is an instantaneous "occurrence of interest", that is, an event occurs at a specific point in time. There is a distinction between *primitive* and *composite events*. Primitive events are elementary occurrences. In the context of databases, primitive events are related to the *modification of data* (e.g. the creation, deletion, or modification of particular objects or of any object in a particular class), to the *retrieval of data* (e.g. the fetch of an object or the invocation of a particular method that retrieves objects), to the *processing of transactions* (e.g. the begin, commit, or abort of transactions), and to *time* (e.g. absolute points in time, such as 7 April 1996, 10.00am, or relative points in time, such as 30 minutes after event X occurred). Data modification, data retrieval, and transaction processing are actions which have a duration. Hence, primitive events can be signalled either *before* the action starts or *after* the action ends. Moreover, events can be signalled from *application programs* and are then called *external events* (also, *abstract* or *explicit events*). Those events are related to occurrences outside the database system and have to be defined and signalled explicitly (e.g. readings from sensors).

Whereas primitive events are single occurrences of interest, composite events represent complex occurrence patterns consisting of a number of primitive events. As such, composite events are expressions built from primitive events and *event operators*, such as *conjunction*, *disjunction*, and *sequence*. For example, "after 31 December 1995, 6.00pm money is debited from the account " depicts a sequence of two primitive events, a time event and a data modification event.

2.1.2 Event Type Versus Event Occurrence

In order to be able to detect primitive and composite events at system runtime, they have to be specified beforehand. An *event language* is used to build composite events from primitive events and event operators. The specification of an event determines its *event type* (synonymous with *event class*). At system runtime, the specified event types are monitored and *event occurrences* (synonymous with *event instances*) are detected. Since an event occurs at a specific point in time, an event occurrence is characterised by a *time of occurrence*. The time of occurrence of a composite event is derived from the times of occurrence of its component primitive events. Typically, it corresponds to the time of occurrence of the primitive event which makes the composite event complete, often called the *terminator event*. Besides the time of occurrence, events may have other *event parameters* describing the circumstances in which the event occurred. In general, an event type determines not only one, but numerous event occurrences which are distinguished on the basis of their event parameters. For example, "customer Smith uses the ATM in Oxford Street, London" can occur numerous times, depending on the behaviour of the customer.

When specifying composite events it may be necessary to impose further restrictions on the possible combinations of primitive events. Those *event restrictions* state conditions on the event parameters, which must be fulfilled at system runtime by the component events of a composite event. Consider a composite event "customer has bonus status ; customer spends more than £100 a month at Tesco" (where ";" represents the sequence event operator) which causes the dispatch of a gift to the customer. The rule makes sense only if the same customer is considered throughout the rule.

2.1.3 Detection of Primitive and Composite Events

The *detection of a primitive event* depends on its kind. There are three techniques for detecting *data modification events*, *data retrieval events*, and *transaction events* [?][1]:

- the *hardwired technique* incorporates code into the implementation of the function.
- the *wrapper-based technique* wraps the function with additional code.
- the *system-supported technique* modifies the function invocation mechanism.

The choice of the technique to be applied depends on the system architecture. *Time events* are detected by programming the timer interface to generate timer interrupts at pre-specified points in time. Finally, *external events* are detected by embedding the code for signalling the event into the application program.

[1]Modifying/retrieving data and processing transactions corresponds to the execution of functions.

The *detection of a composite event* builds upon the detection of primitive events. When a primitive event occurs, it is checked whether it contributes towards the detection of some composite event. If yes, the composite event may be detected or not. In the first case, the composite event is signalled to the system component responsible for rule execution. In the latter case, the primitive event is stored with the composite event. If a primitive event contributes towards the detection of multiple composite events, *rule priorities* determine the order of their evaluation. It is not only important that all primitive events contributing towards a composite event occur, but also that they occur in the correct order. This order is determined by the composite event expression and specifically by the event operators.

The detection process is supported by a *detection mechanism.* This maintains the structure of a composite event and the desired order of primitive event occurrences within it. Each time a primitive event occurs, it is inserted into the detection mechanism. If a step can be performed, the new detection state is derived and the old one is deleted. A certain "final state" indicates the detection of a composite event. In current research projects, the detection mechanism is based on the evaluation of abstractions such as *finite state automata*, *petri nets*, or *trees*. Large numbers of partially detected composite events may arise at the evaluation of such abstractions. Keeping all those events means imposing a storage overhead as well as providing obsolete event data to applications. The *event life-span* determines how long to keep partially detected composite events [**?**].

When detecting composite events, there may be several event occurrences which could satisfy a composite event; for example[2], consider the composite event $E_1 \ ; \ E_2$ and three event occurrences for E_1: e_1^1, e_1^2, and e_1^3. On the occurrence of e_2^1, the *event consumption* must be well-defined, namely, what E_1-event(s) to combine with e_2^1. [**?**] introduces four *parameter contexts*:

Recent combines e_2^1 with the newest event occurrence available (e_1^3, e_2^1) and deletes the consumed events.

Chronicle combines e_2^1 with the oldest event occurrence available (e_1^1, e_2^1) and deletes the consumed events.

Cumulative combines e_2^1 with all event occurrences available $(e_1^1, e_1^2, e_1^3, e_2^1)$ and deletes the consumed events.

[2]In e_m^n, m refers to the event type and n refers to the nth occurrence of an event of that type.

Continuous combines e_2^1 in turn with all event occurrences available $((e_1^1, e_2^1)$-$(e_1^2, e_2^1)(e_1^3, e_2^1))$ and does not delete the consumed events.

2.2 HiPAC

The HiPAC (High Performance ACtive database system) project was started in 1987 with the goal of supporting event-driven applications where timely response to monitored situations is critical [?, ?]. This includes the support of active database capabilities by means of ECA rules. The HiPAC database system is an object-oriented active database system.

2.2.1 Primitive Events

Three kinds of primitive events are supported in HiPAC:

- *data manipulation events*
- *clock events*
- *external notification events*

Data manipulation events are related to the execution of operations on objects. Since the execution of a database operation has a duration, two events can be defined for each operation: the *beginning* of the operation and its *end*. The parameters of data manipulation events are the formal arguments of the operation. Additional environment variables (e.g. the transaction identifier) may be included. Events are also associated with the beginning and the end of transactions. The parameters of such events include the transaction, user, and session identifiers. *Clock events* can be *absolute*, *relative*, or *periodic*. Absolute clock events are signalled when specified time-points are reached (e.g. 23 January 1996, 9:00:00.00). If an event is specified as a temporal offset to some reference event, it is a relative clock event. Relative clock events are signalled when the time interval denoted by the temporal offset has passed after the occurrence of the reference event (e.g. 30 minutes after event X occurred). If an event is to be signalled periodically, it is specified as a periodic event, that is, as a reference event and a time period (e.g. every Friday at 17:00:00.00). *External events* are specified and raised explicitly by users or application programs. When external events are raised the formal parameters are bound to actual values. Examples are readings of sensors, e.g. "a temperature has been reached" or "a person has entered a room".

2.2.2 Composite Events

Three event operators are supported for composite event specification:

- *disjunction* (|)
- *sequence* (;)
- *closure* (*)

A *disjunction* of two events E_1 and E_2, $(E_1 \mid E_2)$, is signalled when either E_1 or E_2 is signalled. The parameters of the disjunction are the parameters of the signalled event. A *sequence* of two events E_1 and E_2, $(E_1 \; ; \; E_2)$, is signalled when E_2 is signalled, provided that E_1 had been signalled before. The parameters of the sequence are the parameters of E_1 and E_2. Finally, the *closure* of an event E_1, $(E_1^\star \; ; \; E_2)$, is signalled when E_1 had been signalled an arbitrary number of times before E_2 is signalled. E_2 serves as a delimitation. The parameters of the closure are the union of all parameters of E_1 events plus the parameters of E_2.

2.2.3 Event Detection

Event detection in HiPAC is not discussed in depth in the available literature. A composite event specification is essentially a regular expression. Hence, composite events can be detected by *finite state automata*, driven by primitive event detectors.

2.3 Ode

Active database facilities in the Ode object-oriented database are described in [**?**, **?**, **?**, **?**]. Two kinds of facilities are provided: *constraints* for maintaining database integrity and *triggers* for automatically performing actions depending on the database state.

2.3.1 Primitive Events

In [**?**] it is mentioned that the set of basic events[3] could be arbitrary in general. Four kinds of basic events are supported in Ode:

- *object state events*

[3]The term *basic event* coincides with the term *primitive event* used in other event specification languages.

- *method execution events*
- *time events*
- *transaction events*

Object state events relate to generic data manipulation operations. An object state event is signalled after an object is created, before it is deleted, and before or after it is updated/read/accessed through a public member function. *Method execution events* are signalled before or after an operation (method) is executed on an object. Similar to clock events in HiPAC, *time events* can be *absolute*, *relative*, or *periodic*. *Transaction events* relate to the beginning and the end of transactions. Such events are signalled after a transaction begins, before or after it commits, and before or after it aborts. Each basic event can be associated with a set of user-defined parameters which carry information about the action that caused the event to occur and/or about the state of the database.

A basic event becomes a *primitive event*, if it is qualified with a mask; that is, the signalling of the event depends on the fulfillment of a condition. The condition may access the event parameters or the state of any object in the database. For example, the basic event "before execution of the withdrawal method" qualified with the mask "withdrawal amount is greater than 1000" filters out all event occurrences with the amount-parameter being less than 1000. In general, Ode supports an EA (Event-Action) model rather than an ECA model, the conditions being merged with the event-part.

2.3.2 Composite Events

The semantics of composite events is explained with respect to an *event history*, which is a finite set of event occurrences in which no two event occurrences have the same event identifier. Intuitively, an event history is the sequence of primitive event occurrences since some starting primitive event.

Four basic event operators are supported for composite event specification:

- *conjunction* ($\wedge$)
- *not* (!)
- *relative*
- *relative+*

A *conjunction* of two events E_1 and E_2, $(E_1 \ \wedge \ E_2)$, is signalled when both events occur at the same event in the event history. Hence, since primitive events cannot occur simultaneously (the event history is a sequence of primitive event occurrences), a conjunction can never be signalled for two primitive events. However, the conjunction can among other things be used to check whether the terminator of a composite event corresponds to some primitive event. A *negation* of an event E, $(!E)$, takes place at all event occurrences in the event history at which E is not signalled. *relative*(E_1, E_2) is signalled when the whole of E_1 occurs before the whole of E_2 (note, that this operator is different to the sequence operator in HiPAC; there, the terminator event of E_2, and not the whole of E_2, must occur after E_1). *relative*+(E) denotes the closure of *relative*(E, E) and is signalled when E or an arbitrary number of successive E's occur. Besides these basic event operators, a number of derived event operators are supported in Ode. For example, a disjunction $(E_1 \ \vee \ E_2)$ and *prior*(E_1, E_2) are derived event operators and have the same semantics as the disjunction and the sequence event operators in HiPAC. A sequence event operator in Ode, *sequence*$(E_1,\ldots,E_m)$, specifies immediate successive occurrences of E_1, E_2, $\ldots$, E_m. Further derived event operators are introduced in the literature.

Composite events inherit parameters from their constituent events. Only the parameters of interest are considered; that is, the user defines which parameters to keep. Moreover, it may be required that certain parameters of constituent events match in a composite event. For example, the composite event *immediate_re_hire(X) = sequence(fire(X), hire(X,Y,Z))* is to be signalled when an employee X is immediately re-hired after being fired. Although *hire* has three attributes, only one attribute is inherited by the composite event *immediate_re_hire*. Moreover, the values for X in *fire* and *hire* must match.

2.3.3 Event Detection

Composite event expressions are equivalent to regular expressions. Hence, composite events can be detected by *finite state automata*. The event history provides the sequence of input symbols to the automaton. The event occurrences are fed into the automaton one at a time, in the order of their event identifiers. The current marking of an automaton determines the current stage of the detection process. If the automaton enters an accepting state, then an event corresponding to the automaton occurred at the last input event. The construction of the finite state automaton M_E of a composite event expression E is discussed in [**?**].

The algorithms shown do not consider the handling of event parameters. Considering event parameters implies providing additional storage and computing new event parameters when the detection progresses. Consider the finite state automaton for *relative*$(E_1,\ E_2)$ shown in Figure **??**, and the event history $\langle e_1^1, e_1^2, e_2^1, e_2^2 \rangle$. It is expected that the composite event is signalled twice, once after e_2^1 and once after e_2^2. However, the finite state automaton is non-deterministic and different outcomes are possible. The reason for this is, that it is not clear when and if the empty-event-transition ε is performed (i.e., after e_1^1, after e_1^2, or not at all).

2.4 SAMOS

The SAMOS (Swiss Active Mechanism-based Object-oriented database System) project addresses the specification of active behaviour and its internal processing [**?**, **?**, **?**, **?**].

2.4.1 Primitive Events

Primitive events relate to occurrences in the database, in the database management system, or in the database environment. The following kinds of primitive events are supported in SAMOS:

- *method events*
- *value events*
- *transaction events*
- *time events*
- *abstract events*

Method events relate to the execution of methods on objects and are signalled before or after the execution. Method events include events concerning generic object operations, which can be regarded as special methods. The parameters of method events are the formal arguments of the method, the object identifier of the object executing the method, and *environment parameters*: the time of occurrence, the occurring transaction (identifier of the transaction during which the event occurred), and the user identifier (identifier of the user who started the occurring transaction).

Value events are associated with the modification of objects, i.e. of object attributes, and are signalled before or after modification. Object attributes can be modified by different methods. Therefore, a value event can be specified as a disjunction of method events. The parameters of a value event are the object identifier, the value of the updated object, and the environment parameters. In case the event is signalled before modification, the object value parameter corresponds to the old object. In case the event is signalled after modification, this parameter corresponds to the new/updated object. *Transaction events* signal the beginning, the commit, and the abort of transactions. The parameters of a transaction event are the environment parameters. *Time events* can be specified as *absolute*, *relative*, or *periodic* and have no parameters. *Abstract events* occur in the environment of the database system and are signalled by users or application programs. Abstract events have to be defined by users, including their formal parameters. These user-defined parameters are instantiated by the user or application program when the event is raised. In addition, the environment parameters are set by the database system.

2.4.2 Composite Events

Six event operators are supported for composite event specification:

- *conjunction* (,)
- *sequence* (;)
- *disjunction* (|)
- $\star$/*last*
- *TIMES*
- *NOT*

A *conjunction* of two events E_1 and E_2, (E_1 , E_2), is signalled when both events E_1 and E_2 have occurred, regardless of their order. The parameters of the conjunction are the parameters of E_1 and E_2. There is only one *environment parameter* for composite events, the time of occurrence. The other environment parameters, occurring transaction and user identifier, are not applicable because constituent events may originate in different transactions started by different users. The time of occurrence parameter of a conjunction corresponds to the time of occurrence of the later event. A *sequence* of two events E_1 and E_2, (E_1 ; E_2), is signalled when E_2 is signalled, provided that E_1 had been signalled before. The parameters of the sequence are the parameters of E_1 and E_2, plus the time of occurrence of E_2. A *disjunction* of two events E_1 and E_2, $(E_1 \mid E_2)$, occurs when either E_1 or E_2 is signalled. The parameters of this occurrence become the parameters of the disjunction (including the time of occurrence). The $\star$- and *last*- event operators are used whenever an event is to be signalled only once during a predefined monitoring interval. $\star$*E IN I* refers to the first occurrence of E in the interval I and *last(E) IN I* to the last occurrence. The parameters of the first and the last occurrence of E respectively become the parameters of the composite event. A composite event *TIMES(n, E) IN I* is employed whenever n occurrences of an event E in the interval I are to be signalled. The parameters of the composite event are the union of all parameters of the n occurrences of E. The time of occurrence is set to the value of the last (nth) time of occurrence. *NOT E IN I* denotes the non-occurrence of an event E in the interval I. A *NOT* event is signalled at the end of the interval I. It has no parameters, except for the time of occurrence which corresponds to the time of occurrence of the event indicating the end of the interval.

A *monitoring interval I* is a time interval during which an event has to occur. For example, *E IN [s - e]* denotes an event *E* which has to occur in the time interval starting with *s* and ending with *e*. *E IN [s - e]* is equivalent to *(TS; NOT TE); E*, where *TS* is the time event representing *s* and *TE* is the time event representing *e*. *(TS; NOT TE); E* expresses that *E* has to occur after *TS*, but before *TE*.

In many cases, it is required that certain parameters of constituent events match in a composite event. Consider, for example, a conjunction between two method events where the corresponding methods have to be executed on the same object. The relationship between the two object identifier parameters can be established by using the *same* keyword. Therefore, *(E_1,E_2):same object* denotes the corresponding event. In general, the *same* keyword can be used to establish the equality between particular parameter values for the constituent events of a composite event.

2.4.3 Event Detection

SAMOS uses *Coloured Petri Nets*, so-called *SAMOS Petri Nets*, for the detection of composite events. A Petri Net consists of *places*, *transitions*, and *arcs*. Arcs connect places with transitions and transitions with places. The places of a Petri Net correspond, intuitively, to the potential (distributed) states of the Net, and such states may be changed by the transitions. Transitions correspond to the possible events which may occur (perhaps concurrently). In Coloured Petri Nets, tokens are of specific *token types* and may carry complex information. Concerning SAMOS Petri Nets, tokens represent event occurrences and capture the event type and the event parameters. A place in a SAMOS Petri Net contains tokens of one specific token type. At runtime, an event occurs and a corresponding token is inserted into all places representing its event type. The flow of tokens through the Net is then determined; a transition can *fire* if all its input places contain at least one token. Firing a transition means removing one token from each input place and inserting one token into each output place. The parameters corresponding to the token type of the output place are derived at that time[4]. Certain output places are marked as *end places*, symbolising specified composite events. Inserting a token into an end place corresponds to the detection of a specified composite event. The event parameters are part of the token.

The SAMOS Petri Net for the composite event $E_1\ ;\ E_2$ is shown in Figure **??**. Place H is an auxiliary place and contains one token before event detection begins. On the occurrence of an event $e_1 \in E_1$, a corresponding token (including event type and event parameters) is inserted into place E_1. Both input places of transition t_1 contain one token and t_1 fires, that is, one output token (including e_1's parameters) is inserted into E_1'. On the occurrence of an event $e_2 \in E_2$, a corresponding token is inserted into place E_2 and transition t_3 fires. In this case, one token (including e_1's and e_2's parameters) is inserted into the end place and hence, the composite event $e_1\ ;\ e_2$ is detected. If an e_2 event is signalled without a previous e_1 event, the token representing e_2 is deleted.

2.5 Sentinel

The Sentinel project is a follow-on project to HiPAC [**?**, **?**]. A major part of the project is the development of Snoop, a model independent[5] event specification language, and of algorithms for the detection of Snoop event expressions [**?**, **?**].

2.5.1 Primitive Events

Three kinds of primitive events are supported in Sentinel:

[4]This is the default behaviour of a Petri Net. More complex behaviours using more than one token from specific input places or checking certain conditions on the set of input tokens can be modelled.

[5]that is, independent of the underlying data model

- *database events*
- *temporal events*
- *explicit events*

Database events are related to database operations, including transactions. At least two events (*begin-of* and *end-of*) can be associated with each database operation. *Temporal events* are divided into two sub-classes: *absolute* and *relative* temporal events. Absolute temporal events in Sentinel include absolute and periodic time events as found in other event specification languages. For example, $< (17 : 00 : 00) * / * /* >$ denotes the periodic time event "every day at 5pm". Finally, *explicit events* depict events which are not part of the Sentinel event language, but are defined by users or application programs. Two parameters, the event type and the time of occurrence, are defined for all primitive events. Other parameters are event specific. Other parameters for a database event depend on the event modifier *begin-of* or *end-of*. Typically, parameters of a *begin-of* event include the input parameters whereas the parameters of an *end-of* event include both, the input and output parameters of the operation. Temporal events have no other parameters.

2.5.2 Composite Events

Six event operators are supported for composite event specification:

- *disjunction* ($\vee$)
- *conjunction* (*Any*)
- *sequence* (;)
- *aperiodic event operators* (A, $A^\star$)
- *periodic event operators* (P, $P^\star$)
- *not* ($\neg$)

A *disjunction* of two events E_1 and E_2, $(E_1 \mid E_2)$, occurs when either E_1 or E_2 is signalled. A *conjunction* event, denoted $Any(m, E_1, \ldots, E_n)$ where $m \leq n$, is signalled when any m events out of the n distinct events specified occur, ignoring the order of their occurrence. A *sequence* of two events E_1 and E_2, $(E_1 \; ; \; E_2)$, occurs when E_2 occurs provided that E_1 has already occurred. An *aperiodic* event $A(E_1, E_2, E_3)$ is signalled when E_2 occurs in the interval formed by E_1 and E_3. This non-cumulative variation of the aperiodic event can occur zero or more times. On the other hand, $A^\star(E_1, E_2, E_3)$, the cumulative variation of the aperiodic event, occurs only once when E_3 occurs and accumulates the occurrences of E_2 (that is, collects the corresponding parameters) in the interval formed by E_1 and E_3. A *periodic* event is an event that repeats itself within a constant and finite amount of time. The event $P(E_1, [t], E_3)$, where E_1 and E_3 are events and t is a time specification, is signalled at all multiples of the time period t within the interval formed by E_1 and E_3. The cumulative variation of the periodic event $P^\star(E_1, [t], E_3)$ allows a parameter specification to be attached to t. The parameters are accumulated until the end of the interval (the occurrence of E_3). For example, $P^\star(8am, [30min] : \textit{IBM-stock-price}, 5pm)$ samples the IBM stock every $30min$ from $8am$ to $5pm$. Finally, the *not* operator $\neg E_2[E_1, E_3]$ detects the non-occurrence of the event E_2 in the closed interval formed by E_1 and E_3.

The parameter computation for composite events is as follows. The parameters for a conjunction and a sequence are the union of the parameters of the participating events. $A(E_1, E_2, E_3)$'s parameters are the parameters of E_2 plus the event type, whereas these parameters are accumulated until the end of the interval for $A^\star(E_1, E_2, E_3)$. $P(E_1, [t], E_3)$ has only two parameters, the event type and the time of occurrence. For $P^\star(E_1, [t], E_3)$, the parameter specification attached to t determines which parameters are to be accumulated.

The parameter contexts *recent*, *chronicle*, *cumulative*, and *continuous* (see Section 2.1.3) were first introduced in connection with the Sentinel research project [**?**]. With each ECA rule a parameter context is specified, which determines the evaluation semantics for the event-part.

2.5.3 Event Detection

Sentinel uses *trees* for the detection of composite events. Each composite event is represented as an *event tree*. Event trees are then coalesced to an *event graph* (that is, common subtrees are shared among different event trees) in order to avoid the redundant evaluation of composite events. The leaves of an event graph represent primitive events and the nodes represent composite events. A node is marked with the event operator and its children correspond to the event operands of a composite event. The event operands are either nodes (that is, composite events) or leaves (that is, primitive events). At runtime, primitive events occur and are injected into the leaves corresponding to their event type. The leaves pass the primitive events directly to their parent nodes. An activated parent node executes a procedure which evaluates the incoming data (depending on the event operator and the parameter context). If the corresponding composite event is detected, it is propagated to the parent node. If the corresponding composite event is not detected, the incoming event data is either stored temporarily in a list of subevents or it is disregarded. In the first case, the event can be used later on, at the arrival of other suitable events. In the latter case, the event cannot be used. Nodes marked with a rule identifier correspond to specified composite events. Composite events detected at such nodes are signalled to the *condition evaluator*.

Figure **??** shows the event tree for the composite event $E_1 \; ; \; E_2$. When an event $e_1 \in E_1$ is signalled, it is injected into the leaf representing E_1 and propagated to the parent node, where it is appended to E_1's list. When an event $e_2 \in E_2$ is signalled, it is also propagated to the parent node. The further evaluation depends on whether there are elements in the list of E_1's subevents or not. If there are no elements, e_2 is disregarded. If there are elements, one element of the list is combined with e_2 and the composite event is detected. Which of the elements in E_1's list is chosen depends on the parameter context.

2.6 Other Research Projects

The research projects addressed so far consider object-oriented active database systems. There is a number of research projects considering relational active database systems. Three of those projects, *POSTGRES*, *Ariel*, and *Starburst*, each include the development of an active database rule language and its complete implementation on top of existing relational database systems.

2.6.1 POSTGRES

The POSTGRES project was carried out at the University of California in Berkeley [**?**, **?**]. The event-part of ECA rules has the form "on *event* to *object*", where *event* can be any operation caused by a POSTQUEL command[6]: retrieve, replace, delete, or append, and *object* can be either a relation or a column of a relation. These primitive events are detected using *event locks*: an *object* is tagged with the *event*.

[6]POSTQUEL is the query language of POSTGRES.

When an appropriate event occurs on a field tagged with an event lock, the primitive event is detected. Composite events are not supported in POSTGRES.

2.6.2 Ariel

Events in Ariel [?] are similar to events in POSTGRES, that is, relate to retrieve-, replace-, delete-, or append-operations on relations or columns of a relation. In addition, Ariel supports absolute and periodic *time events*. Composite events are not allowed.

2.6.3 Starburst

In Starburst [?, ?], primitive events are related to insertions into tables, deletions from tables, and updates to tables or table columns. Simple composite events can be expressed using a *disjunction* event operator. Rule processing is invoked at the end of *transitions*, which are non-empty sequences of insert-, delete-, and update-operations. The overall *effect of a transition* is captured in *transition tables*, one for insertions, one for deletions, and one for updates. Rule processing means that transition tables are evaluated. If a specified event (or events, in the case of a disjunction) is contained, a rule is triggered.

2.7 Events in Distributed Active Database Systems

A common feature of all research projects discussed in the previous sections is that they consider centralised active database systems. Sentinel mentions some problems of ECA rules in distributed environments, does, however, not solve them.

2.7.1 Implications in Distributed Computing Environments

The differences between centralised and distributed ECA rule processing are due to the special characteristics of distributed systems: concurrent processes running at multiple autonomous sites, no global time, message delays between sites, problem of global state, and independent failure modes (see Section 3.1).

Composite events in distributed systems can involve component events at many sites. These system-wide composite events are called *global composite events*, as opposed to *local composite events* which relate to event occurrences at a single site and coincide with composite events discussed earlier in this chapter. Each global composite event is monitored at one specific *observer site* which contains the *global* *event detector*. Global event detectors of different global composite events are distributed to arbitrary sites. The main implication of the special characteristics of distributed systems on global composite event detection is:

> The order in which events are signalled at global event detectors does, in general, **not** correspond to the order in which the events occurred.

The contrary is assumed in all "centralised-system" research projects studied in the previous sections; if an event e_2 is signalled at the event detector later than an event e_1, e_2 occurred after e_1. Therefore, the detection of global composite events must be based on the time of occurrence parameters of component events. However, the second implication of the special characteristics of distributed systems on global composite event detection is:

> Time of occurrence parameters originating at different sites are inaccurate.

This implication represents a further complication for the detection of global composite events. As a result, the detection of global composite events is considerably different from the detection of local composite events.

2.7.2 Related Work

In [**?**], Jagadish and Shmueli consider composite events in object-oriented databases. Event processing in centralised and distributed computing environments is compared and the particular difficulties of distributed event processing are highlighted. The notions of *s-correctness* and *computational correctness* are introduced in order to achieve a centralised event processing behaviour and different ways for obtaining s-correctness and computational correctness are presented. [**?**] considers the order in which events are posted to objects from either one (in the case of a centralised database system) or multiple (in the case of a distributed database system) distributors. In "real" distributed systems s-correctness, and computational correctness respectively, can only be achieved by using atomic broadcast communication protocols [**?**]. This is, however, a far-reaching assumption which imposes a high performance overhead, especially in wide area networks.

Rule processing in general is discussed by Ceri and Widom in [**?**]. This work also aims to achieve a behaviour corresponding to a centralised computing environment. A hierarchy of paradigms for distributed rule processing is introduced. A particular level in this hierarchy determines the expressiveness of rules reaching from totally localised rules to rules which allow *reading and modifying remote data*, *intersite priorities* and *autonomous rule processing at each site.* A "centralised" behaviour is achieved through locking schemes; the more powerful the rule processing level, the more complex the locking scheme. However, Ceri and Widom do not consider global composite events. Instead, conditions and events relate to local data and operations on that data.

[**?**] also discusses rule processing, although this work concentrates on the condition-part of ECA rules. Evaluating a condition corresponds to distributed query processing. The problem of *global state* is addressed and a method for evaluating global conditions using *simultaneous regions* is presented. The motivating idea of this work is to distribute rule processing in order to increase performance. A similar approach is taken in [**?**], which also deals with conflict analysis for the parallel execution of rules.

2.8 Summary

This chapter discussed events in active database systems. The different research prototypes may be characterised by their means for specifying and detecting events. Basically, the relational active database systems POSTGRES, Ariel, and Starburst support only simple primitive events, whereas the object-oriented active database systems HiPAC, Ode, SAMOS, and Sentinel support complex primitive and composite events. In the latter, there is agreement on the different kinds of primitive events. However, composite event specification and detection mechanisms differ considerably. HiPAC laid the foundation-stone for active database research, considering ECA rules in general, but it provides only basic mechanisms for composite event specification and detection. Ode introduces powerful event operators. However, their number and notation is confusing and their application sometimes unclear (i.e. conjunction and negation). Moreover, event detection with finite state automata is non-deterministic and ignores event parameters. SAMOS and Sentinel are projects that emphasise composite event specification and detection. Both are well thought out. In SAMOS, the evaluation of events is captured in SAMOS Petri Nets. Although this ensures precise semantics, it does not reflect the original purpose of Petri Nets as a modelling and analysis tool for concurrent systems. In Sentinel, the event operators are different from other research projects and one needs to become familiar with them; using trees for the detection of composite events is straightforward. SAMOS composite event specification and detection facilities have been implemented in their entirety. In the case of Sentinel, this is not clear.

One characteristic common to all research prototypes is that they consider centralised systems only. Little work has been done on distributed systems. However, as distributed database systems are becoming commonplace, it is necessary to reconsider event specification and detection in the light of their special characteristics.

Chapter 3

Events in Distributed Debugging Systems

Software debugging is the process of locating the causes of errors in a software system and suggesting possible repairs [?]. The debugging of distributed software systems is more complicated than the debugging of centralised software systems, due to the special characteristics of distributed systems: concurrent processes running at multiple autonomous sites, no global time, message delays between sites, the problem of establishing global state, and independent failure modes (see Section 3.1). There are two approaches to software debugging: *state-based debugging* and *event-based debugging*. Event-based debugging tools are more appropriate for debugging distributed software systems than state-based debugging tools. *State-based debugging* is the traditional approach, where users guess what the erroneous behaviour is, determine which pieces of state information illustrate this behaviour, and devise plans for obtaining it, e.g. by setting breakpoints. State-based debugging tools rely on time-invariant program execution and the availability of a controllable, accurate global state. However, in distributed systems problems often appear as performance impairment (i.e. inappropriate behaviour) rather than as erroneous state. [?] states that "behaviour is activity that has observable effects in an executing system" (page 2) and that "an event represents the occurrence of significant system behaviour" (page 4). This motivates the second software debugging approach, namely *event-based debugging*. Events are made visible to debugging tools by instrumenting the software system with additional code that signals events. However, on this low level an executing software system produces a vast amount of events. [?, page 16] speaks of 30000 events/minute. Event-based debugging tools allow the user to design high-level behaviour models in terms of events and event operators describing a desired event pattern. At system runtime, the designed behaviour models representing expected system behaviour are compared to the actual system behaviour. Differences between the two illustrate erroneous behaviour.

The goal of this chapter is to study different approaches to event-based debugging in distributed systems. A short background on distributed systems is given in Section 3.1. The most prominent research projects are addressed thereafter. A comparison of the different approaches is given in Section 3.6.

3.1 Introduction to Distributed Systems

In event-based approaches to debugging, a *distributed system* is viewed as a collection of processes running on different processors. Each process emits a sequence of events. Processes do not share memory but communicate via message passing. The processors are allocated at different component computer systems in the distributed system, which are interchangeably called *sites*, *nodes*, or *hosts*. A single site accommodates one or multiple processors, depending on whether it is based on a uniprocessor or multiprocessor architecture [Bac92].

3.1.1 Characteristics of Distributed Systems

Distributed systems are inherently different from centralised systems. The following list summarises the special characteristics of distributed systems that impact distributed debugging:

- *No global time*
  Each site in a distributed system has its own *local clock*. These clocks can drift and therefore record slightly different times.

- *Message delays between sites*
  Messages sent over a computer network can be delayed depending on the load at the sender and receiver sites and the network load.

- *Problem of global state*
  The global state of a distributed system is the union of the states of the individual sites. Since local states are exchanged by sending messages over the computer network, a constructed global state could be obsolete, incomplete, or inconsistent.

- *Independent failure modes*
  The sites and transmission channels of a distributed system may fail independently of each other.

3.1.2 Time and Order Based on Causality

Due to the lack of global time, two events at different processors cannot always be ordered, that is, it cannot be determined which event happened before the other. Hence, the "happened before" relation defines only a *partial ordering.* In [?], Lamport bases the definition of *happened-before* on the causality principle. An event a happened-before an event b, $a \rightarrow b$, if a could have influenced b (i.e. the processor state of b is derived from the processor state of a); a and b are said to be *causally dependent.* If neither $a \rightarrow b$ nor $b \rightarrow a$, the events are said to be *concurrent* (*causally independent*), written $a \,\|\, b$. A system of *logical clocks* is introduced which assigns a natural number to each event. Logical clocks satisfy the *clock condition*: if $a \rightarrow b$, then a's timestamp is smaller than b's timestamp. However, the contrary is not true; if a's timestamp is smaller than b's timestamp, then a and b may be concurrent. Logical time is said to be *consistent with causality*, but does not *characterise causality. Vector clocks* are a generalisation of logical clocks [?]. A vector of natural numbers is assigned to each event, one for each process. The use of *vector timestamps* enables the determination of $\rightarrow$ and $\|$ relationships between events. Hence, vector time characterises causality.

3.2 EBBA

EBBA (Event-Based Behavioural Abstraction) is a high-level approach to debugging complex software. The EBBA project was started in 1982 at the University of Massachusetts in Amherst [?, ?, ?, ?, ?, ?].

3.2.1 The Notion of an Event

Primitive Events Versus High-Level Events

An *event* corresponds to the occurrence of a significant system behaviour. As such, an executing program can be regarded as a sequence of events, called an *event stream.* In a distributed system, the event stream is a merging of the events generated at different sites. There is a distinction between *primitive* and *high-level events.* Primitive events represent non-decomposed fundamental behaviour. For example, the process control part of an operating system would have primitive events such as `create_process` and `suspend_process` and the file I/O subsystem of the same operating system would have primitive events such as `open_file` and `close_file`. On the other hand, high-level events represent user-defined behaviour models that attempt to explain some layered system component or some complex interaction of primitive system elements.

In order to be recognisable by the system, primitive and high-level events have to be defined. The *Event Definition Language (EDL)* allows the definition of *event classes* for primitive and high-level events. A specific individual occurrence of an event class is referred to as an *instance* of that event class. Different instances of the same event class are distinguished by their *attributes*.

The Definition of Primitive Events

The definition of a primitive event class contains the system-wide unique name of the event class and a list of names and types for the event class attributes. For example, the following EDL expression depicts the definition of a primitive event class `e_openFile` representing the successful request to open a file [?]:

```
event
    (e_openFile <process_id> <filename> <fd> <time> <location>)
   attributes
    process_id:  BitString (size4);
    filename:  PrintableString;
    fd:  Integer
end_event
```

The attributes `process_id`, `filename`, etc. capture the circumstances under which a specific event occurs. Two attributes, `time` and `location`, are defined for all events and denote the time when the instance was created and the location where the instance originated respectively. All other attributes are event class specific. The following instance records an occurrence of the corresponding primitive event:

```
(e_openFile 119 ‘‘etc/services’’ 5 14:30:23.49 ‘‘sluggo:xterm’’)
```

The Definition of High-Level Events

High-level events are defined in terms of primitive or other high-level events and *event operators*. The event operators describe the acceptable orderings of constituent events with respect to the event stream. The following event operators are supported for high-level event specification:

- *sequence* ($\bullet$)
- *choice* ($|$)
- *concurrency* ($\triangle$)
- *repetition* ($^+$ or *)

A *sequence* indicates that the right operand event occurs after the left operand event. If the occurrence of one of two operand events is to lead to the detection of a high-level event, the *choice* event operator is applied. The *concurrency* operator expresses that both operand events must occur and may be arbitrarily interleaved in time. As such, this event operator can be understood as a conjunction operator. Finally, the *repetition* depicts that there are one or more repetitions of the operand event (in the case of $^+$) and zero or more repetitions respectively (in the case of *).

The semantics of event operators and especially of the sequence operator is not revealed in the EBBA literature. Although the problems regarding global time and ordering of events in distributed systems are mentioned repeatedly, no specific semantics is defined. However, the following statements indicate the understanding of temporal order:

- "Nothing can be determined regarding the temporal relations of events created on different processors" [**?**, page2].
- "The binary sequence operator specifies that its operand events are to occur in the left to right order in which they are written. ... Implicitly, the events are related by a <-relation on their time attributes." [**?**, pages 10,11]
- "The sequence operator specifies that event instances ... must follow each other in the event stream ... " [**?**, page7]

These statements indicate that the events emitted to the event stream from a particular site arrive at the event recogniser in the same order, namely in the order of their occurrence (this implies FIFO network delivery). Moreover, the sequence event operator can only be applied to event classes originating at the same site.

The following EDL expression denotes the definition of a *behaviour model* `ProcessFile` describing a simple file processing action [**?**]:

```
model ProcessFile(name: PrintableString) =
    e_openFile • e_readFile+ • e_closeFile
  constraints
    e_openFile.name == name;
    e_openFile.fd == e_readFile.fd;
    e_openFile.fd == e_closeFile.fd
  event
    (m_processFile <name> <pid> <time> <location>)
  attributes
    name: PrintableString = name;
    pid: BitString (size4) = e_openFile.process_id
end_model
```

When the event stream matches the behaviour model `ProcessFile`, a high-level event `m_processFile` is created. The parameter `name` is instantiated by the user to focus on specific files. The event expression denotes a sequence of three event classes, `e_openFile`, `e_readFile`, and `e_closeFile`. The middle event may occur more than once. The *constraining clauses* (indicated by `constraints`) contain restrictions on possible event instance combinations. In this case, they ensure that all event instances relate to the same file. The **event**-part describes the event class which is signalled when a corresponding behaviour is detected. The types and bindings of all attributes except `time` and `location` are given in the **attributes**-part. Note, that the `time` attribute corresponds to the time when the instance is created, that is, the time of event detection.

3.2.2 Behaviour Recognition in the EBBA Tool Set

The goal of distributed debugging is to understand the differences between the desired system behaviour and the actual system behaviour as observed by the user in terms of recognised behaviour models. If a user has a good understanding of what the system is doing, the set of behaviour models he is monitoring will match the event stream closely. However, if a user knows what the system is doing, there is no need to debug it. Hence, in most cases the behaviour models will not match the event stream closely. In these cases, the user will want to gather as much relevant information as possible in order to improve his models, that is, in order to get a better understanding of the system. There are two possible techniques: *decomposition* and *partial interpretations.* The first technique means decomposing a behaviour model into smaller units (namely subexpressions) and evaluating the units independently. Matched units are then synthesised to a more-complete behaviour model. Any missing parts represent potential differences which need to be investigated. The second technique means combining the event instances representing components of a high-level event in all possible ways (note, that there may be multiple event instances for a component event), in the hope that at least one combination matches the behaviour model.

An implementation of the EBBA approach is provided by the *EBBA tool set*. The tool set functions are organised into the following areas:

Model-building and maintenance tools are employed for compiling EDL definitions of primitive and high-level events and distributing them to other tool set components.

Model abstraction tools accept requests for recognition of behaviour models representing primitive and high-level events, evaluate the event instances arriving on the event stream, and signal detected high-level events.

Event collection and communication tools are used for receiving event instances from the event stream and feeding them to the model abstraction tools (for event recognising sites) or emitting generated primitive events to the event stream (for event generating sites).

Interface tools present a graphics-based user interface to the other tools.

At the beginning of a debugging session, a user requests that the event recogniser monitors the event stream for occurrences of specific behaviour models. Those behaviour models are then retrieved and displayed to the user. If necessary, the user supplies actual parameters for the behaviour model heading (such as the parameter `name` in the `ProcessFile` behaviour model). Finally, the user specifies a set of actions to be carried out when the event represented by the behaviour model is recognised. For example, the user might specify that an event instance representing the behaviour model just recognised is created and added to the event stream.

The formalism used to recognise high-level behaviour models is similar to *finite state automata*. However, the input symbols to a so-called *shuffle automaton* are complex n-tuples that resemble events. Also, the input stream is not evaluated one symbol at a time, but transitions are based on sets of symbols, called *transition sets*. *Transition descriptors* describe the transitions between the states of a shuffle automaton in terms of transition sets. There is no detailed description of the structure of shuffle automata and of their evaluation rules in the literature.

3.2.3 Integration with the Target System

The EBBA tool set can be used for *remote* as well as for *distributed debugging.* Remote debugging is implemented by placing the user and the debugging tools at a single site of the distributed system. This central debugging site may or may not participate in the distributed computation. Each remote site contains an agent in order to support the central debugging site. The drawbacks of remote debugging are:

- the large overhead on network traffic, since all detected primitive events are sent un-filtered to the central debugging site.
- the latency of high-level event detection, which implies the display of out-of-date information to the user and the delay of intervention activities.

Distributed debugging is implemented by partitioning and replicating the debugging tools at multiple sites; the model building and maintenance tools run at the user's site. The drawbacks of remote debugging are eliminated in distributed debugging.

3.3 Global Breakpoints

Haban et al. discuss the possibility of using so-called *global breakpoints* in distributed programs [?, ?, ?]. If a global breakpoint is reached, the distributed program is halted.

3.3.1 The Specification of Global Events

Primitive events describe significant behaviour of the execution or the state of a single process. As such, a process can be modelled as a sequence of primitive events and a distributed system can be modelled as parallel streams of events. Note, that this understanding of a distributed system differs from that in Section 3.2; there, a distributed system is modelled as a single event stream, merged from the event streams of different sites. Primitive events are associated with statements of a programming language. For example, primitive events may be associated with processes (such as `start_process` and `stop_process`), with message transfer (such as `send_message` and `receive_message`), or with program execution (such as `enter_procedure` and `leave_procedure`). One or more parameters, including a *timestamp* parameter, are specified with each primitive event class to capture the circumstances under which the event occurs. The timestamp denotes the time of an event occurrence.

Global events are composed of primitive events and/or other global events and *event operators.* Global breakpoints are defined in terms of global events. The timestamp of a global event corresponds to the timestamp of the last primitive event participating in its occurrence. Apart from this, no information about parameters of global events is given in the literature. The following event operators are supported for global event specification:

- *alternation* (|)
- *conjunction* (&)
- *happened-before* (<)
- *simultaneity* (^)
- *negation/between* (@)

An *alternation* $A_e = Ge_1 \,|\, Ge_2$ is detected when either of the operand events occurs. The timestamp of that occurrence is inherited by A_e. If the occurrence of both operand events is to lead to the detection of a global event, the *conjunction* operator is applied. The timestamp of $C_e = Ge_1 \& Ge_2$ corresponds to the timestamp of the later operand event. The *happened-before* event $H_e = Ge_1 < Ge_2$ indicates that Ge_1 happened before Ge_2. The timestamp of H_e is the timestamp of Ge_2. As opposed to the happened-before event, the operand events of a *simultaneous* event cannot be ordered, but occur simultaneously. The event $S_e = Ge_1^{\wedge} Ge_2$ inherits the timestamp of the last operand event which caused the satisfaction of S_e. Finally, the negation operator can be applied in two different ways. First, the *negation* event $N_e = Ge_1 @ Ge_2$ depicts that Ge_1 occurs and that no Ge_2 occurred previously. N_e has Ge_1's timestamp. Second, the *between* event $B_e = @Ge_3(Ge_1, Ge_2)$ expresses that no Ge_3 event occurs between the start of an interval (Ge_1) and the end of an interval (Ge_2). The timestamp of B_e corresponds to the timestamp of Ge_2.

In a debugging system, a causal relationship is essential in order to locate the cause of errors. Therefore, time relationships between events are based on causality (see Section 3.1.2). The timestamps of events correspond to vector time. For a primitive event, the timestamp denotes the time of event occurrence and is read from the logical clock at the occurrence site. Logical clocks tick after the occurrence of a primitive event. For a global event, the timestamp is derived from the constituent events. The system of logical clocks builds a *logical global time base.* [**?**] examines the semantics of event operators based on the vector timestamps of operand events. The semantics of $<$ and $^\wedge$ corresponds to the semantics of Lamport's happened-before $\rightarrow$ and concurrency $\|$.

3.3.2 The Detection of Global Events

A *local debugger* is associated with each node in the distributed system. The local debuggers are connected to a *central test station CTS*. CTS coordinates the debugging process and represents the user interface.

A global event class is represented as a tree structure, called a *breakpoint tree*. Inner nodes are labelled with event operators and leaves are labelled with primitive or global events. Each node of a breakpoint tree contains a list for event instances satisfying the corresponding event class. At the beginning of a debugging session, the breakpoint trees are constructed on CTS and distributed to all local debuggers which are involved in the corresponding global event.

During the debugging session, primitive events occur and are signalled at all local debuggers. Their vector timestamps are then inserted into all leaves corresponding to their event class. As a result, the attached operator nodes are evaluated taking into account the events already stored in the event lists. If no corresponding event is detected, the primitive event is stored in the event list. If a corresponding event is detected, it is propagated to the parent node and the procedure recurses or, if there is no parent node, the global event is detected. In this case, corresponding event instances are deleted from the breakpoint tree and the detected event instance is signalled to all local debuggers. If the global event denotes a global breakpoint, CTS is informed, the distributed system is halted, and corresponding information is displayed to the user.

3.3.3 Realisation of the Debugging System

There are two approaches for implementing global breakpoints: a *pure software approach* and a *hardware supported approach*. In the first case, the local debuggers and the CTS share the same execution environment as the system under analysis. This implies interference with the processors and additional load on network traffic due to the transmission of events. The disadvantage of the software approach is that interference with the program being debugged can alter its runtime behaviour. In the second case, the local debuggers and the CTS run on separate hardware, connected by a separate network. Hence, there is only a little overhead on the system under analysis due to the generation of events.

3.4 Data Path Debugging

This work [?] is an extension of Bruegge and Hibbard's *generalised path expressions* for debugging sequential programs [?]. There, a *path rule* is an EA (event-action) rule, where the event-part denotes a generalised path expression, that is an expression built from *path functions*[1] and the operators *repetition* (*), *sequencing* (;), and *exclusive selection* (+). Predicates can be associated with any path function. Basic path expressions (generalised path expressions without predicates) correspond to regular expressions and can therefore be recognised by *finite state automata.*

3.4.1 Events in Data Path Expressions

Data path expressions (DPEs) are employed for debugging parallel programs. There are four kinds of events:

- *control events*
- *data events*
- *compound events*
- *conditional events*

Control events represent control activities, such as entering or leaving a procedure. *Data events* denote conditions and occur when the condition becomes true. For example, "$[X = X' + 1]$" is the event that X is incremented (X' refers to the previous value of X). Note, that data events can only be detected by checking the current program state regularly. Hence, the detection of data events is relatively inefficient. DPEs are called *compound events* and they are defined as "event_id = dpe". Finally, *conditional events* denote any of the previous three kinds of events with attached predicates. Their format is "event [condition]", and they are detected when the event occurs while the condition holds.

3.4.2 The DPE Hierarchy

DPEs are classified into five subclasses, depending on their syntactic structure and the corresponding semantic model:

- *sequential DPEs*

[1]Path functions correspond to single statements of a programming language; that is, they are equivalent to primitive events.

Subclass	Sequential DPEs	Multiple DPEs	Safe DPEs	General DPEs	Extended DPEs
Expresses	sequential behaviour	limited safe concurrency	safe concurrency	limited unsafe concurrency	unsafe concurrency
Syntax	dpe_1: EVENT | (dpe_1) | $\text{dpe}_1 + \text{dpe}_1$ | $\text{dpe}_1 ; \text{dpe}_1$ | $\text{dpe}_1^{\star}$	dpe_2: dpe_1 | dpe_2 & dpe_1	dpe_3: EVENT | (dpe_3) | $\text{dpe}_3 + \text{dpe}_3$ | $\text{dpe}_3 ; \text{dpe}_3$ | $\text{dpe}_3^{\star}$ | dpe_3 & dpe_3	dpe_4: dpe_3 | $\text{dpe}_4 + \text{dpe}_4$ | dpe_4 & dpe_4 | dpe_4 @	dpe_5: EVENT | (dpe_5) | $\text{dpe}_5 + \text{dpe}_5$ | $\text{dpe}_5 ; \text{dpe}_5$ | $\text{dpe}_5^{\star}$ | dpe_5 & dpe_5 | dpe_5 @
Semantics	finite state automata	k-safe nets (subset)	k-safe nets	Petri Nets	extended Petri Nets (subset)

Table 3.1: DPE Hierarchy

- *multiple DPEs*
- *safe DPEs*
- *general DPEs*
- *extended DPEs*

Table 3.1 illustrates the syntax and the semantics of the DPE hierarchy [?, page 13]. The &-operator expresses that both operands occur causally independent and the @-operator depicts the closure of & ($\varepsilon + A + A\&A + A\&A\&A + \ldots$).

Sequential DPEs correspond to generalised path expressions. The second subclass, *multiple DPEs*, allows only global-level concurrency but no nested concurrency. A multiple DPE can be detected by a subclass of Petri Nets, called *k-safe nets*, where the maximum number of tokens in a place is limited to k. *Safe DPEs* denote safe concurrency, that is, bounded parallelism, and are equivalent to k-safe nets. The fourth subclass, *general DPEs*, allows unbounded parallelism, but unbounded parallel threads never join (as in $A@; B$). This subclass is equivalent to Petri Nets. Finally, *extended DPEs* express unsafe concurrency and can be detected by extended Petri Nets.

3.4.3 Predecessor Automata

Finite state automata are efficient recognisers for sequential behaviour, but they cannot represent concurrent events that are causally independent. [?] introduces *pre-*

decessor automata which extend finite state automata in the appropriate way. The authors claim that Petri Nets, which can represent sequential as well as concurrent behaviour, are not efficient for recognising events at runtime and that predecessor automata overcome this deficiency. Predecessor automata can recognise behaviour with safe concurrency and can therefore implement safe DPEs, the third subclass of the DPE hierarchy. [**?**, page 171] mentions that predecessor automata can become quite complex. The definition of a predecessor automaton is the same as the definition of a finite state automaton, except that the transition function determines a new successor state depending on an event plus information about its predecessors. On the occurrence of an event, a transition can only be made if the corresponding predecessors have occurred as well.

Figure **??** shows the predecessor automaton for $((A; C)\&B); D$. A labelled edge corresponds to a transition. For example, $(a\ .)$ depicts the occurrence of an event a without any predecessors and $(d\ c\ b)$ denotes the occurrence of an event d with two predecessors, c and b. One possible input to the automaton is $(a\ .)$ $(b\ .)$ $(c\ a)$ $(d\ c\ b)$, which corresponds to the state sequence $1 - 2 - 5 - 6 -$ *final state*.

Since the order in which events are received at the event recogniser is different from their order of occurrence, events have to be ordered before entering the predecessor automaton regarding their (partial) occurrence order in the system under analysis.

3.5 Other Research Projects

3.5.1 EVEREST

EVEREST (EVEnt REcognition teSTbed) is a system to analyse the behaviour of distributed computations [**?**]. The goal of EVEREST is to study different approaches to event recognition by applying different *monitoring strategies*. The background of primitive and high-level event definition and detection is similar to Global Breakpoints (see Section 3.3). However, *flexible structuring* allows the user to specify the monitoring configuration of the system, that is, the types of monitoring modules to be created, their placement in the system, and the distribution of event recognition modules. The *separation of event and monitoring definition* makes it possible to analyse the same high-level events under different monitoring configurations. Moreover, different monitoring configurations can be utilised concurrently. Another useful feature is *dynamic configuration*, namely, the possibility to change the monitoring configuration dynamically at system runtime. Finally, the existence of *multiple time view protocols* gives the opportunity to specify different evaluation semantics for high-level event detection. For example, the semantics of the happened-before and simultaneity event operators can be based on vector time, as in Section 3.3, or on local physical time. The existence of global physical time is assumed in the latter case, which implies that timestamps originating at different nodes are compared without considering any clock inaccuracies.

The motivation for the EVEREST project arose from earlier work on reducing the delay between the occurrence of an event and its detection, and the *nature of events* in general [**?**, **?**].

3.5.2 Monitoring Distributed Systems

In [**?**, **?**], Mansouri-Samani and Sloman are concerned with the monitoring of distributed systems. Monitoring is needed for debugging, testing, program visualisation and animation, and for the management of distributed systems and communication networks in general. In particular, the authors address numerous problems related to the implementation of monitoring tools in object-based distributed systems, such as the conflict between the concept of encapsulation and event-driven monitoring. A *monitoring model* is defined in terms of four monitoring functions.

- **Generation of monitoring information** means the detection of primitive events and the regular generation of status reports. The objects concerned must be instrumented appropriately with software/hardware probes or sensors. The primitive events and status reports may be recorded as monitoring traces for later analysis.

- **Processing of monitoring information** indicates that the raw and low-level monitoring data is converted to the required format and level of detail. Note, that distributed systems generate large volumes of monitoring data. The options for processing include (1) increasing the level of abstraction by employing high-level events (2) filtering monitoring information and (3) merging traces in order to provide a global view.

- **Dissemination of monitoring information** denotes that generated monitoring reports are forwarded to users. For example, a monitoring report may be broadcast to all users or its dissemination may depend on a subscription scheme.

- **Presentation to the user** expresses that received monitoring information is displayed to the user in a suitable form. Taking into account that monitoring information can be vast, it is important that the user can specify a specific display.

On the whole, the contribution of this work lies in the provision of a reference model for explaining different approaches to monitoring distributed systems.

3.6 Comparison

Table 3.2 compares the features of EBBA, Global Breakpoints, and safe DPEs, the third subclass in the Data Path Debugging hierarchy.

In EBBA, the concurrency operator ($\triangle$) is employed for events at one processor as well as for events at different processors and expresses an interleaving semantics[2], similar to a conjunction. Hence the expressiveness of EBBA is limited, because two causally dependent events cannot be distinguished from two causally independent events. Another characteristic of EBBA is that events signalled from a single processor have to be ordered before entering a shuffle automaton. This could be achieved by implementing FIFO network delivery.

The Global Breakpoints syntax allows causal dependence and causal independence between events to be expressed. The existence of the negation operator @ further increases the expressiveness of the event language. However, the fact that there may be multiple event instances of the same event class leads to semantic difficulties in connection with the negation operator. Depending on which event instance is consumed in a specific case, the outcome of an evaluation can differ, that is, a global event may either be detected or not detected.

Finally, safe DPEs can also express causal dependence and causal independence between events. The use of predecessor automata for event detection implies that all events carry explicit information about their predecessors. This means that causal dependence is not detected in the predecessor automaton, but in the source processor. In the same way as for EBBA, events have to be ordered before entering a predecessor automaton.

[2]The operand events may be arbitrarily interleaved in time.

Systems	EBBA	Global Breakpoints	Data Path Debugging
Simple Events	primitive events	primitive events	path functions
Complex Events	high-level events	global events	safe DPEs
– Conjunction	concurrency $\triangle$	conjunction &	–
– Disjunction	choice |	alternation |	exclusive selection +
– Sequence	sequence $\bullet$	happened-before <	sequencing ;
– Closure	repetition $^+$ and *	–	repetition *
– Concurrency	concurrency $\triangle$	simultaneity $^\wedge$	causal independence &
– Negation	–	negation/between @	–
Order	physical order (within a process)	causal order	causal order
Timestamp	absolute (time of detection)	vector (time of occurrence)	– –
Detection Mechanism	shuffle automata	breakpoint trees	predecessor automata
Recognition	remote or distributed	distributed (with replication)	centralised

Table 3.2: Distributed Debugging Systems – A Comparison

3.7 Summary

This chapter described the use of events in distributed debugging systems. The different research prototypes agree on what constitute primitive events and the construction of composite events; Global Breakpoints provides additional expressiveness through the negation event operator. However, the semantics of event detection differs considerably in the different research prototypes. EBBA has weak semantics, allowing only the concurrency event operator $\triangle$ to be applied to events at different sites. On the other hand, EBBA's debugging tools are remarkably sophisticated. In Global Breakpoints and safe DPEs, the semantics is based on causal order (see Section 3.1.2). The first uses vector time to determine causal order. Vector time imposes a scalability problem, because a vector timestamp contains one time value for each site. In safe DPEs, the causal order is explicit in primitive event occurrences, that is, a primitive event occurrence contains information about which primitive event occurrences preceded it. It is not clear how such primitive event occurrences can be detected generally.

Chapter 4

Analysis

4.1 Review of Related Work

In active database and distributed debugging systems, events are employed for monitoring the runtime behaviour. On the detection of a pre-specified primitive or composite event, the system reacts accordingly. Table 4.1 summarises the main features of both research areas, addressed in Chapters 2 and 3 respectively. The main observation is that current active database systems monitor the external and internal behaviour of centralised systems (by employing time events regarding physical time), whereas distributed debugging systems monitor the internal behaviour of distributed systems (by monitoring the cause-effect relationship).

The following list summarises the positive (+) and negative (−) features of active database systems:

+ expressive event languages (different kinds of primitive events, negation event operator, other derived event operators)

+ convenient handling of event parameters (parameters pre-defined and instantiated by the system for all events except abstract events)

+ well-defined and clear semantics, including event consumption

− only centralised systems, not directly extendible to distributed systems

In active database systems, the specification and detection of primitive and composite events is highly developed. All aspects of how to specify a behaviour pattern, including the state information to be retrieved (in the form of accumulated event parameters), and of how to detect this behaviour pattern at system runtime are well

Active Database Systems	Distributed Debugging Systems
consider centralised systems • arrival order at event detector $=$ occurrence order • no problem with physical time	consider distributed systems • arrival order at event detector $\neq$ occurrence order • problem with physical time
monitor external and internal system behaviour • real time • time events • temporal order	monitor internal system behaviour • no real time • no time events • causal order
primitive and composite events • no concurrency event operator	primitive and composite events • concurrency event operator
semantic model • determination of event ordering: uncomplicated • well-defined event consumption • rule priorities	semantic model • determination of event ordering: complicated (no agreement) • undefined event consumption • no rule priorities
centralised event detectors	centralised, distributed, or replicated event detectors

Table 4.1: Features of Active Database and Distributed Debugging Systems

thought out and semantically sound. However, to date active database research has focussed on centralised systems. This will not be satisfactory in the future, since interest is moving all the time towards distributed database systems.

The following list summarises the positive (+) and negative (−) features of distributed debugging systems:

+ clear event languages (similar to regular expressions)

+ attempt to minimise interference with normal runtime behaviour

+ powerful monitoring tools and sophisticated user interfaces

− handling of event parameters is unclear

− fuzzy semantics and undefined event consumption

The main concern of distributed debugging systems is to master the vast amount of primitive event occurrences in distributed computations without interfering with normal runtime behaviour[1]; the user specifies an expected runtime behaviour and the distributed debugging system compares the specified runtime behaviour with the actual runtime behaviour. Since the user has only a vague idea of the actual runtime behaviour, the specification is necessarily somewhat fuzzy. Hence, the fuzzy semantics is intentional to some extent. However, in Chapter 3 it was shown that some negative features are due to omissions in the various research projects.

4.2 Goals of This Work

The goals of this work are:

- to develop algorithms for monitoring the external and internal behaviour of distributed (database) systems
- to give a clear semantics for the specification and detection of primitive and composite events
- to employ fully distributed event detectors
- to reflect upon the special characteristics of distributed systems
- to support event parameter handling
- to realise the algorithms using some distributed programming system

In summary, the work on active database systems is to be applied to distributed systems, taking into account the existing work on distributed debugging systems. The results show how to *monitor the behaviour of distributed systems* in general.

[1]Interference can lead to different runtime behaviour, i.e. an error is caused by the debugger itself.

Chapter 5

Syntax of Events

Chapter 2 discussed events in active database systems and Chapter 3 focussed on events in distributed debugging systems. The goal of this chapter is to present a specification language for primitive and composite events which has the expressiveness of event languages in active database systems, but extended to accommodate distributed environments. The following sections address the syntax of primitive events, the syntax of composite events, and the notion of event parameters respectively. An extensive example illustrating the specification of primitive and composite events for a telecommunication application follows in Section 5.4.

5.1 Primitive Events

Active database systems support different kinds of primitive events concerning occurrences inside a database system (data manipulation events and transaction events) and in its environment (time events and abstract events). On the other hand, distributed debugging systems relate primitive events to the statements of a programming language. In this thesis, primitive events should have the complexity of primitive events in (centralised) active database systems including time events for expressing physical time, and should be applicable in distributed systems.

Definition 5.1.1 In a distributed system, *primitive event types are site-related*, that is, events at different sites are considered as of different type.

There are numerous reasons for relating primitive event types to occurrences at single sites:

- Two primitive events at a single site are semantically different from two primitive events at remote sites; their evaluation differs and therefore they should be considered separately.
- The *naming* of primitive event types imposes a problem in large-scale (possibly heterogeneous) distributed systems. Site-relative naming provides both scaling and federation.
- In order to implement specified primitive and composite event types, an interest in primitive event types has to be *registered* with corresponding sites. Primitive event occurrences are then *notified* at the registered event detectors [?]. Site-information is necessary for the registration-notification process.

Definition 5.1.2 A *primitive event expression* is a character string denoting either a time event, a data manipulation event, a transaction event, or an abstract event. A specified primitive event expression determines a *primitive event type.*

Notation The site of origin of a primitive event type is made explicit with `site_n` $\triangleright$ *primitive event expression.*

The following sections discuss the different kinds of primitive events and specifically, their application in distributed systems. Primitive events in distributed debugging systems fall into the category of abstract events.

5.1.1 Time Events

Naturally, since an event is an isolated instant in time [?], any reached point in time represents a primitive event. In a distributed system, points in time are designated with respect to the readings of local clocks. Hence, a specified time event denotes the readings of a particular local clock. Time events can be *absolute*, *relative*, or *periodic.* An *absolute time event* is given in terms of a date and time specification and the site of a local clock, `site_n` $\triangleright$ `day/month/year:-hours:minutes:seconds.fraction`. The granularity of `fraction` is based on the clock resolution (e.g. milliseconds). *Relative time events* define a temporal offset to some reference event, such as `(event X + 30 minutes)`. Since the time period (`30 minutes` in this case) is added to the time of occurrence of the reference event (`event X` in this case), the time period should be measured at the site where the reference event occurred. Hence, the example above corresponds to `site_of_occurrence(X)` $\triangleright$ `(event X + 30 minutes)`. Finally, *periodic time events* identify recurring time events, such as `site_n ▷ every Friday at 17:00:00.00`. Again, `site_n` identifies the site of a local clock.

5.1.2 Data Manipulation Events

Data is manipulated in terms of database operations. In relational database systems, these are the *insertion*, *deletion*, and *modification* of tuples. In object-oriented database systems, these are *method executions* in general. Since database operations have a duration between their beginning and their end, two corresponding primitive events can be identified: the *beginning* of an operation (referred to as `BEFORE`) and its *end* (referred to as `AFTER`). See Chapter 2 for an in depth discussion of data manipulation events in different research prototypes. In distributed systems, the specification of a data manipulation event must include a site specification, if a corresponding data manipulation operation exists at different sites. For example, consider an object class `accounts` which is distributed to different sites (corresponding to the branches of a bank), the event `site_n ▷ AFTER.withdraw(account)` identifies the end of a `withdraw`-operation on `account`-objects at `site_n`. Occurrences of `AFTER.withdraw(account)` events at arbitrary sites can be specified as a disjunction of primitive events. Henceforth, `AFTER` is considered as default and may be omitted in the specification.

5.1.3 Transaction Events

A *distributed transaction* consists of a number of *subtransactions.* One site coordinates the distributed transaction, the *distributed transaction manager (DTM).* Each site employs a *local transaction manager (LTM)* for dealing with the subtransactions at that site [?].

As in centralised database systems, the *begin*, the *commit* and the *abort* of a distributed transaction (signalled by the DTM) denote primitive events (referred to as `BOT`, `EOT`, and `ABORT` respectively). In addition, each subtransaction issues a *local_begin* and a *local_commit* or *local_abort* (referred to as `l_BOT`, `l_EOT`, and `l_ABORT` respectively). Furthermore, the creation of a subtransaction denotes a primitive event, either the *remote_create* of the DTM (referred to as `r_COT`) or the *local_create* of the LTM (referred to as `l_COT`). Again, the specification of a transaction event includes a site specification, such as in `site_n ▷ BOT`.

5.1.4 Abstract Events

Abstract events are events which are signalled from outside a database system, either by users or by application programs. Consequently, abstract events include primitive events in distributed debugging systems, where a primitive event can be associated with any statement of the programming language. The names of abstract event types have to be defined explicitly with `DEFINE PRIM_EVENT event_name`. The system-wide unique naming of abstract events is the responsibility of the user, that is, he/she has to ensure that no two distinct abstract event types carry the same name.

5.2 Composite Events

Composite events are made up of primitive and/or other composite events and *event operators*. Different kinds of primitive events and the specification of primitive event expressions were discussed in the previous section. This section deals with event operators. The kinds of event operators applied in active database systems and in distributed debugging systems are similar, except that there is no concurrency event operator in active database systems, because they are centralised. In distributed systems, it is necessary to introduce a concurrency event operator in order to express that two events at different sites occur "concurrently". But, what does "concurrently" exactly mean? In distributed debugging systems, the notions of "concurrency" and "happened before" are based on causality (see Section 3.1.2). This is not feasible when dealing with physical time. An in depth discussion of the semantics of composite events and specifically, of the notions of "concurrency" and "happened before", will be given in Chapter 6.

The set of event operators should build a representative cross-section of the event operators used in active database systems and in distributed debugging systems. The event operators should be expressive enough in order to specify even complex behaviour patterns as composite event expressions. However, in the context of this thesis it is not necessary to introduce a wide variety of event operators which make the specification of events more convenient without increasing the expressive power of the event specification language.

Definition 5.2.1 A *composite event expression* is defined as follows:

- A primitive event expression is a composite event expression.

- If E_1, E_2, and E_3 are composite event expressions, then $(E_1\ ,\ E_2)$, $(E_1 \mid E_2)$, $(E_1\ ;\ E_2)$, $(E_1 \parallel E_2)$, $(E_1\ ^\star\ E_2)$, $(E_1\ ^+\ E_2)$, and $(E_1\ ;\ \text{NOT}\ E_2\ ;\ E_3)$ are composite event expressions.

- Nothing else is a composite event expression.

Composite event types are either defined explicitly with

```
DEFINE EVENT event_name = composite_event_expression
```

or implicitly, in the ON-part of an ECA rule definition [?], with

```
DEFINE RULE rule_name =
    ON  composite_event_expression
    IF  < condition >
    DO  < action >
```

In the first case, event_name determines the name of a composite event type. This name can be reused in the definition of other composite event types. In the latter case, ON designates the event-part, IF the condition-part, and DO the action-part of an ECA rule. The event-part ON can either be a composite event expression or any name of a composite event type previously defined.

The (,) event operator denotes a *conjunction* and (|) a *disjunction*. The notions of "happened before" and "concurrency" are captured in the *sequence* (;) and the *concurrency* (‖) event operators. Finally, (*) and ($^+$) depict *iterations* and (NOT) a *negation*.

Note, that not all composite event expressions are meaningful. For example, the composite event expression `DEFINE EVENT Neg =` E_1`; NOT` E_2`; (`$E_2\ ^+\ E_3$`)` can never lead to the detection of an event, since E_2 is required in the occurrence of ($E_2\ ^+\ E_3$) and at the same time forbidden in NOT E_2. [?] discusses this issue in the context of active database systems and [?] in the context of distributed debugging systems. The latter gives a formal model for detecting unreasonable composite event expressions.

Definition 5.2.2 A *local (composite) event expression* is a composite event expression whose constituent event expressions relate solely to one site[1].

Definition 5.2.3 A *global (composite) event expression* is a composite event expression whose constituent event expressions relate to more than one site.

[1]Note, that all primitive event expressions are local event expressions.

It is convenient to distinguish local and global event expressions. All constituent events of a local event occur at the same site. Hence, the special characteristics of distributed systems do not have any influence on the semantics and the detection of local events. Instead, the same results as in (centralised) active database systems apply to local events in distributed systems.

An event is an isolated instant in time [**?**], that is, an event occurs at a specific point in time. For primitive events, the *time of occurrence* corresponds to the time of the local clock just after event detection (except for time events, where the time of occurrence corresponds to the event itself). For composite events, the time of occurrence is derived from the times of occurrence of the primitive events participating in that occurrence. Since a composite event is detected at the occurrence of the "last" such primitive event, the *terminator event* (see Section 2.1), the time of occurrence of that event becomes the time of occurrence of the composite event. In distributed systems, it is not only important to know when an event occurred, but also where it occurred. Hence, an event has a *location of occurrence* which corresponds to the origin of the time of occurrence. Time of occurrence and location of occurrence together build the *timestamp* of an event. Composite events are detected on the basis of the timestamps of their constituent events.

5.2.1 Conjunction: $Ce = (E_1\ ,\ E_2)$

The conjunction-operator (,) is applied, if both operand events are to occur and may be arbitrarily interleaved in time. E_1 and E_2 may originate at the same or at different sites. The timestamp of Ce corresponds to the timestamp of the "later" event of E_1 and E_2. The meaning of "later" depends on whether Ce denotes a local or a global event and has to be examined carefully.

5.2.2 Disjunction: $De = (E_1 \mid E_2)$

There are two possible semantics for the disjunction-operator (|): *exclusive-or* and *inclusive-or*. Applying exclusive-or means detecting De as soon as one of E_1 or E_2 occurs, whereas inclusive-or considers both operand events if they occur "at the same time". In centralised systems, no two events can occur "at the same time" and hence, the disjunction-operator always corresponds to exclusive-or. In distributed systems, two events at different sites can occur "at the same time" and hence, both exclusive-or and inclusive-or are applicable. The timestamp of De corresponds to either E_1's or E_2's timestamp, if only one operand event led to the occurrence of

De. Otherwise, it is derived from the timestamps of both operand events.

5.2.3 Sequence: $Se = (E_1 \ ; \ E_2)$

The sequence denotes that event E_1 "happens before" event E_2. However, the semantics of "happens before" differs, depending on whether *Se* is a local or a global event. Therefore, although the syntax is the same for local and for global events, the two cases have to be considered separately. The timestamp of *Se* is the timestamp of the later event E_2.

5.2.4 Concurrency: $Pe = (E_1 \parallel E_2)$

The concurrency-operator ($\parallel$) is used, if both operand events are to occur virtually "at the same time". This implies that this operator applied to two distinct operand events[2] is only applicable in global events; the operand events E_1 and E_2 occur at different sites and it is not possible to establish an order between them. The timestamp of *Pe*[3] is derived from the timestamps of both operand events.

5.2.5 Iteration: $Ie_1 = (E_1 \ ^{\star} \ E_2)$ and $Ie_2 = (E_1 \ ^{+} \ E_2)$

The iteration-operators are applied, if all occurrences of E_1 events are to be collected before the occurrence of an E_2 event. E_2 serves as a delimitation, that is, Ie_1 and Ie_2 occur when E_2 occurs and inherit the timestamp of E_2. * implies that there are zero or more occurrences of E_1 before E_2 for the detection of Ie_1, whereas there are one or more occurrences of E_1 for the detection of Ie_2.

5.2.6 Negation: $Ne = (E_1 \ ; \ NOT \ E_2 \ ; \ E_3)$

A negation event corresponds to a restricted sequence event, where the sequence $(E_1 \ ; \ E_3)$ is only signalled, if there was no occurrence of E_2 in between. Hence, the timestamp of *Ne* is the timestamp of E_3. In some cases, it is desirable that the monitoring period starts immediately, that is, that the E_1 event operand is omitted. This is achieved by substituting E_1 with the special time event *Now* (see also [**?**]) which is signalled at the beginning of an event monitoring session.

The negation event *Ne* can be represented as an event with a *monitoring interval*: $E_3 \ IN \, [E_1 \ldots E_2]$; the event E_3 has to occur in the time interval starting with E_1 and

[2]Note, that an event occurs at the same time as itself.

[3]*Pe* stands for *Parallel event*.

ending with E_2. This notation is convenient and will be used later in this chapter.

5.3 Event Parameters

Detected event occurrences have parameters, which capture the circumstances under which the event occurred. In active database systems, the parameters of a detected event are used to evaluate the condition and to execute the action of an ECA rule. In distributed debugging systems, the parameters are used to display debugging information to the user.

In the previous section, it was mentioned that all events, primitive and composite, have a *timestamp* parameter. The timestamp indicates when and where the event occurred. Other parameters of primitive and composite events are discussed subsequently.

5.3.1 Primitive Event Parameters

Each primitive event occurrence has a timestamp parameter. Since primitive event types are site-related (see Proposition 5.1.1), the location of occurrence (which is part of a timestamp) is inherent in the event type. However, as will be shown in the next chapter, a timestamp is meaningless without information on its origin and hence, the location of occurrence is always stated explicitly. [?] introduces two other *environment parameters* for primitive events: the *occurring transaction*, which identifies the transaction during which the event occurred, and the *user identifier*, which identifies the user who started the occurring transaction.

Time Event Parameters

Time events have only a timestamp parameter. For absolute time events, the value of this parameter is inherent in the specified event. For relative and for periodic time events, it is calculated at system runtime in order to set timer interrupts for signalling the events. The timer interrupt for a relative time event is set after the reference event occurred and the timer interrupt for a periodic event is set periodically, for example, after the last interrupt was generated.

Data Manipulation Event Parameters

Besides a timestamp, data manipulation events have the environment parameters occurring transaction and user identifier. The other parameters correspond to the

Composite Event	Timestamp	Other Parameters
$(E_1\ ,\ E_2)$	MAX$\{E_1, E_2\}$	parameters of E_1 and parameters of E_2
$(E_1 \mid E_2)$ exclusive-or	E_1 or E_2	parameters of E_1 or parameters of E_2
$(E_1 \mid E_2)$ inclusive-or	E_1 and/or E_2	parameters of E_1 and/or parameters of E_2
$(E_1\ ;\ E_2)$	E_2	parameters of E_1 and parameters of E_2
$(E_1 \parallel E_2)$	E_1 and E_2	parameters of E_1 and parameters of E_2
$(E_1 \ {}^{\star}\ E_2)$	E_2	union of parameters of E_1 and parameters of E_2
$(E_1 \ {}^{+}\ E_2)$	E_2	union of parameters of E_1 and parameters of E_2
$(E_1\ ;\ \text{NOT}\ E_2\ ;\ E_3)$	E_3	parameters of E_1 and parameters of E_3

Table 5.1: Parameters of Composite Events

formal arguments of a data manipulation operation and the relation/object to which the operation applies. The parameters of data manipulation events are instantiated by the database system.

Transaction Event Parameters

Transaction events have a timestamp and the other environment parameters occurring transaction and user identifier.

Abstract Event Parameters

Besides a timestamp, the parameters of abstract events are user-defined with `DEFINE PRIM_EVENT event_name(param1,param2,...)`. The user-defined parameters are instantiated explicitly by the user or application program, when the abstract event is raised. This is not true for data manipulation events, where the parameters are instantiated by the database system.

5.3.2 Composite Event Parameters

The parameters of a composite event are derived from the parameters of its constituent events. As mentioned earlier, each composite event has a timestamp parameter, which indicates when and where the event occurred. The timestamp parameter was addressed informally in the previous section. A formal discussion within the context of the general semantics of composite events will follow in Chapter 6. Table 5.1 gives an overview of the parameters of composite events depending on the event operator.

5.3.3 Parameter Restrictions

In many cases, it is necessary to impose further restrictions on the possible combinations of events. Those *parameter restrictions* state conditions on the event parameters, which must be fulfilled at system runtime by the constituent events of a composite event (see Section 2.1). For example, it may be necessary to relate all constituent events of a composite event to the same object or to the same transaction. Parameter restrictions are handled differently in different research prototypes; Ode (Section 2.3) and EBBA (Section 3.2) permit arbitrary relational expressions between event parameters, SAMOS (Section 2.4) permits equality between event parameters, and Sentinel (Section 2.5) disallows parameter restrictions completely. In Sentinel the justification is that parameter restrictions are conditions and that conditions should not be merged with events, but rather appear in the condition-parts of ECA rules [**?**, page 30]. However, consider the example "customer has bonus status ; customer spends more than £100 a month at Tesco". Without parameter restrictions the event detector would combine all "customer has bonus status"-events with all "customer spends more than £100 a month at Tesco"-events, regardless of the values for customer. A large number of composite events with non-matching customers would be detected and disregarded in the condition-part.

In this thesis, specifying the equality of event parameters is supported. For example, `DEFINE EVENT` Com$_1$`(param) = `E$_1$`(param) event_op `E$_2$`(param)` states that the value of `param` in an occurrence of E$_1$ should be equal to the value of `param` in an occurrence of E$_2$, and that the common value should be stored as `param` with the other parameters of a detected composite event. The definition of a composite event includes only the parameters concerning parameter restrictions. Other parameters are derived automatically (see Section 5.3.2). Storing `param` and its value explicitly with the parameters of a composite event makes it possible to specify parameter restrictions for more complex composite events, such as `DEFINE EVENT` Com$_2$`(param) = `Com$_1$`(param) event_op `E$_3$`(param)`. There are three possibilities for using parameter restrictions with a negation:

`DEFINE EVENT Neg = `E$_1$`(param) ; NOT `E$_2$`(param) ; `E$_3$ This case is slightly peculiar in that it does not restrict the possible combinations of events; an E$_1$ event is *not* combined with an E$_2$ event. Rather, the `param` parameter of E$_2$ is bound by the earlier occurrence of E$_1$ and E$_2$ events with parameters unequal `param` are allowed to occur in between E$_1$ and E$_3$. Since the sequence event E$_1$ `;` E$_3$ can have arbitrary parameter combinations, no additional parameter is derived for the composite event.

`DEFINE EVENT Neg(param) = E`$_1$`(param) ; NOT E`$_2$` ; E`$_3$`(param)` This case is similar to an ordinary sequence event with parameter restrictions; `param` must match in `E`$_1$ and `E`$_3$ and no `E`$_2$ (with arbitrary parameters) occurs in between.

`DEFINE EVENT Neg(param) = E`$_1$`(param) ; NOT E`$_2$`(param) ; E`$_3$`(param)` This case represents a combination of the previous two cases.

Note, that it is not sensible to impose parameter restrictions on a composite event specified with a disjunction.

5.4 Examples

Telecommunication applications are inherently distributed. The scenario outlined in this section deals specifically with the world of *telephoning*. Each company, almost each household in the western world, and a considerable number of households in developing countries are connected to the telephone network. An interconnection between any two telephone subscribers can be established in a course of seconds. A number of computers for switching telephone calls are involved in this interconnection process.

Figure **??** shows the North American Telephone Switching Office Hierarchy for interregional telephone calls between any two subscribers A and B in regions served by different regional centres [**?**]. Alternate routing is available. Solid lines connecting switching offices indicate circuits that are always available, provided calling volume is not too great. Dotted lines indicate circuits installed only when the amount of traffic between different switching offices merits installation. For the case illustrated, a maximum of ten switching offices can be used for one telephone call. The five levels of switching offices may be supplemented by one or two more for international telephone calls.

A number of telephone companies jointly provide the North American telephone network. Telephone subscribers rent their telephones from one specific company, but may have accounts with other companies as well. In this way, the cheapest rates for local, national, and international telephone calls can be exploited.

Example 5.4.1 Three major telephone companies, AT&T, GTE, and BELL, offer a special service to customers having an account with each of them and paying their bills “promptly”. The definition of “promptly” differs between different telephone companies[4].

```
DEFINE EVENT prompt_pay_AT&T(customer) =
   payment_AT&T(customer) IN
   [invoice_AT&T(customer)...invoice_AT&T(customer) + 7 days]
DEFINE EVENT prompt_pay_GTE(customer) =
   payment_GTE(customer) IN
   [invoice_GTE(customer)...invoice_GTE(customer) + 14 days]
DEFINE EVENT prompt_pay_BELL(customer) =
   payment_BELL(customer) IN
   [invoice_BELL(customer)...invoice_BELL(customer) + 10 days]
DEFINE RULE Conjunction_Example =
   ON prompt_pay_AT&T(customer) , prompt_pay_GTE(customer) ,
      prompt_pay_BELL(customer)
   IF true
   DO special service(customer)
```

Three `prompt_pay`-events are defined depending on the telephone company. If a `customer` pays his/her bill within a certain period from invoicing (7 days for `AT&T`, 14 days for `GTE`, and 10 days for `BELL`), a `prompt_pay`-event is signalled. Different `payment`- and `invoice`-event types may occur at different sites.

The `ON`-part of the rule defines a conjunction between the three `prompt_pay`-events with the same parameter values for `customer`. Again, the `prompt_pay`-events occur at different sites, namely at the clearing-houses of the corresponding telephone companies.

Example 5.4.2 In North America, telephone fraud is an immense problem costing telephone companies millions of dollars each year. One possibility to detect telephone fraud is to monitor the use of PINs (Personal Identification Numbers). A PIN can only be used by its owner and therefore the concurrent use of a single PIN at two different locations indicates an irregularity and is reported.

```
DEFINE EVENT make_phone_call =
   site_1 ▷ make_phone_call |...| site_n ▷ make_phone_call
DEFINE RULE Concurrency_Example =
   ON make_phone_call(PIN) || make_phone_call(PIN)
   IF timestamp₁ ≠ timestamp₂
   DO inform authorities(PIN)
```

[4]In the following, the name of an example rule connotes the event operator which is illustrated.

Telephone calls for a specific PIN can be made from a large number of telephones and are registered at one of n registration sites. Hence, the event `make_phone_call` represents a disjunction (exclusive-or) of primitive `make_phone_call`-events, one for each of these sites (note, that primitive event types are site-related).

A concurrency between two different events implies that the events occur at different sites. However, an event is concurrent with itself (reflexivity). Therefore, it is demanded that the `make_phone_call`-events have different timestamps, that is, denote different events. The `PIN` parameter values must be the same for both events.

Example 5.4.3 Another possibility to detect telephone fraud is to monitor average usage patterns each day. The following rule expresses that all telephone calls for a specific PIN on a specific day are collected and evaluated. If the most expensive telephone call exceeds \$100 or the total amount exceeds \$400, the authorities are informed.

```
DEFINE RULE Iteration_Example =
   ON make_phone_call(PIN)+ IN site_i ▷ EVERY DAY [0.00...24.00]
   IF MAX{amount} > $100 OR SUM{amount} > $400
   DO inform authorities(PIN)
```

All `make_phone_call`-events (defined as in Example 5.4.2) for a specific `PIN` occurring within a day are collected. The events may occur at different sites. The periodic time events `EVERY DAY 0.00` and `EVERY DAY 24.00` relate to the local clock at `site_i`, which may be the site where the composite event is detected. The `amount`-parameter-values of the collected `make_phone_call`-events are then evaluated in the `IF`-part.

Example 5.4.4 Telephone customers can help detecting telephone fraud; going on holiday they deregister their PIN. If the PIN is used thereafter, before being reregistered, the authorities are informed.

```
DEFINE RULE Sequence1_Example =
   ON make_phone_call(PIN) IN [deregister(PIN)...reregister(PIN)]
   IF true
   DO inform authorities(PIN)
```

Again, the events `deregister`, `reregister`, and `make_phone_call` (defined as in Example 5.4.2) may occur at different sites.

Example 5.4.5 Telephone customers should settle their telephone bills within a month of invoicing. If this is not the case, the telephone company issues a reminder.

```
DEFINE RULE Negation_Example =
   ON invoice(tele_com, customer) ; NOT payment(tele_com, customer) ;
      (invoice(tele_com, customer) + 1 month)
   IF true
   DO send reminder(customer)
```

After the occurrence of the `invoice`-event, the parameter values for `tele_com` and `customer` are fixed. That means, if a `payment`-event for this parameter tuple does not occur within a month from invoicing, the `NOT`-event occurs and the action is executed.

Example 5.4.6 Routing telephone calls from Local Central Offices to International Offices should cause as little overhead as necessary. For that reason, standard routes are checked regularly in order to detect the fastest option. A signal is broadcast from some Local Central Office to two (or more) International Offices, IC_1 and IC_2 in this example. The signal received first determines the future routing.

```
DEFINE RULE Sequence2_Example =
   ON signal_received_IC1 ; signal_received_IC2
   IF true
   DO set standard route to IC1
DEFINE RULE Sequence3_Example =
   ON signal_received_IC2 ; signal_received_IC1
   IF true
   DO set standard route to IC2
```

The `signal_received`-events occur at different sites, namely at the International Offices IC_1 and IC_2. This example shows that it is important to establish the order between two events from different sites to a fine granularity.

5.5 Summary

This chapter discussed the specification of primitive and composite events in distributed systems, including the treatment of event parameters. The resulting event specification language derives from earlier work on (centralised) active database systems and distributed debugging systems. An extensive example illustrating the specification of primitive and composite events for a telecommunication application was presented.

Chapter 6

Semantics of Events

In the previous chapter, the specification of primitive and composite events in distributed systems was discussed. The goal of this chapter is to identify the formal semantics of primitive and composite events, that is, to identify when and where an event occurs. The semantics of primitive events is straightforward: a primitive event occurs and its timestamp is allocated when the occurrence is detected. The semantics of composite events is more subtle; it depends on the timestamps of constituent primitive and composite events and on event operators. The formal semantics of composite events is developed in the following sections. First, the notions of time and order in distributed systems are investigated. In this dissertation, the notions of time and order relate to physical time and temporal order. The structure and handling of timestamps are discussed thereafter. Section 6.3 addresses the general semantic model and Section 6.4 addresses the simplified semantic model. The latter is applicable in specific distributed systems only. Finally, the formal semantics of events is presented in Section 6.5.

6.1 The Notion of Physical Time in Distributed Systems

A fundamental result of Einstein's Special Theory of Relativity is that there is no absolute physical time; *time is relative* to the observer. Depending on the frame of reference, the relative order between two events may be reversed for two different observers. Although the influence of special relativity is negligible in common distributed systems, it is essential for understanding the notion of time as a whole; *there is a physical limitation to the ordering of events.*

The notion of physical time is also a problem in distributed systems; there is *no global time.* Each site of a distributed system contains at least one local clock (depending on the processor architecture, see Section 6.2.1). These clocks are electronic devices that count oscillations occurring in a crystal at a definite frequency. Subsequently, this count will be referred to as *local clock tick.* Some software device reads local clock ticks and calculates the corresponding date and time-of-day, subsequently called the *local time.* Depending on the underlying oscillators and the environment temperature local clocks drift apart. In order to assess event occurrences with respect to the system environment and the happenings at remote sites, it is necessary to *synchronise local clocks.* Special time servers build the basis for clock synchronisation. These time servers are equipped with receivers, for example GPS (Global Positioning System) [**?**], for catching UTC (Universal Time Coordinated) signals from land-based radio stations or satellites. UTC signals have an accuracy in the order of $0.1 - 10$ milliseconds. The time servers transmit messages containing time information to the remote sites of a distributed system.

There are different protocols for clock synchronisation, for example NTP (Network Time Protocol) [**?**], which is the standard for clock synchronisation in the Internet. Typically, NTP achieves a synchronisation error of below 100 milliseconds even in wide area networks (WANs). For a detailed discussion on synchronisation protocols, see [**?**].

6.2 Time and Order

In this thesis, time is modelled as a *discrete* quantity, that is, the timeline is a sequence of discrete ticks isomorphic to the natural numbers. This accords with the definition of an event being an isolated instant in time.

6.2.1 Centralised Systems

Centralised systems are based on *uniprocessor* or *multiprocessor* architectures. Each processor has its own local clock. In a multiprocessor system, one processor may be allocated for timer readings and interrupts [Bac92]. Hereafter, a *single local clock* per centralised system is assumed for detecting time events and allocating timestamps. If this were not the case, a distributed system would be considered on a per-processor rather than a per-site basis.

Depending on the local clock resolution two successive events may correspond to the same timestamp. In order to distinguish such events and determine their order, a counter variable could be employed which is reset to 0 each time the local clock ticks and incremented with each event occurrence. Timestamps could then be extended with an additional field capturing its value.

In a centralised system with a single local clock, two distinct events e_1 and e_2 can always be ordered by comparing their (extended) timestamps; either e_1 happened before e_2, or e_2 happened before e_1. Hence, events are *totally ordered.*

The following list summarises the assumptions for centralised systems:

- There is a single local clock for detecting time events and allocating timestamps.
- Events are totally ordered. Distinct events have distinct timestamps.

6.2.2 Distributed Systems

In distributed debugging systems, the causal relationship is essential in order to locate the cause of errors. Hence, the notions of time and order are based on *logical clocks* and *causal order* (see Chapter 3). However, in this thesis the notions of time and order have to be based on *physical clocks* and *temporal order.* A corresponding model is developed in this section.

Each site in a distributed system has a single local clock. The considerations in the previous section are valid for each of these sites. In order to establish the temporal relation between two events at distinct sites, it is necessary to compare timestamps allocated at these sites. Hence, timestamps need to be globally meaningful. In a distributed system with synchronised local clocks this is best achieved through an *approximated global time base* [**?**, **?**]. Conceptually, there is a *reference clock z* with *granularity* g_z. The synchronised local clocks are characterised by their *precision* Π, which is the maximum time difference between two corresponding ticks of any two local clocks.

Definition 6.2.1 (Precision) Let *$clock_z(clock_k(i))$* 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:

$$\Pi = Max \; \{ \; \forall i \; \forall k \; \forall l \; : \mid clock_z(clock_k(i)) - clock_z(clock_l(i)) \mid \}$$

A global time can be approximated by adjusting the granularity of local clock measurements to a *global clock granularity* g_g. If g_g was less than Π, two simultaneous events might receive timestamps distant more than one clock tick. This would not be meaningful with respect to establishing the temporal relation between two events at distinct sites. Consequently g_g should not be smaller than Π ($g_g > \Pi$). This is known as the *granularity condition.* Figure **??** shows corresponding ticks of three local clocks as measured by the reference clock.

Assuming $g_g > \Pi$, two simultaneous events receive timestamps distant at most one clock tick. The reason for this is that corresponding g_g-intervals of local clocks are shifted against each other by up to Π. Figure **??** shows two event occurrences on two distinct sites. Site$_k$'s clock is faster than Site$_l$'s clock with a drift of Π between them. Although e_1 happens before e_2 with respect to the reference clock, their timestamps imply an inverse ordering ($e_1 : 3$ and $e_2 : 2$). Hence, if two events at distinct sites are less than two clock ticks apart, their temporal relation cannot be deduced from their timestamps and they are said to happen *concurrently.* If the distance is at least two clock ticks, their temporal relation can be determined, since $2g_g > g_g + \Pi$. The ordering of remote events is called *$2g_g$-precedence.*

Definition 6.2.2 ($\mathbf{2g_g}$-restricted temporal order) The global clock granularity g_g is given. *$2g_g$-restricted temporal order* $\rightarrow_{2g_g}$ between primitive events e_1 and e_2 is defined as follows:

1. If e_1 and e_2 are primitive events occurring at the same site and e_1's timestamp is smaller than e_2's, then $e_1 \rightarrow_{2g_g} e_2$.
2. If e_1 and e_2 are primitive events occurring at distinct sites and e_1's timestamp is at least $2g_g$ smaller than e_2's, then $e_1 \rightarrow_{2g_g} e_2$.

Definition 6.2.3 ($\mathbf{2g_g}$-restricted concurrency) The global clock granularity g_g is given. *$2g_g$-restricted concurrency* $\|_{2g_g}$ between primitive events e_1 and e_2 is defined as follows:

$$e_1 \|_{2g_g} e_2 \text{ iff } \neg(e_1 \rightarrow_{2g_g} e_2) \text{ and } \neg(e_2 \rightarrow_{2g_g} e_1)$$

$2g_g$-restricted temporal order defines a *partial ordering* on primitive events in distributed systems, whereas events in centralised systems are totally ordered. It should be noted that the best ordering can be achieved, if g_g is just greater than Π ($g_g = \Pi + \varepsilon$).

Example 6.2.4 The global clock granularity $g_g = 1/100sec$ is given. Consider two primitive events e_1 and e_2.

Event	*Site*	*Global Time*	*Local Clock Tick*
e_1	k	19/10/95,2:32:27.32	23991548127
e_2	?	19/10/95,2:32:27.33	23991548334

If ? has the value k, the events originate at a single site and their local representatives are compared in order to determine the temporal order $e_1 \rightarrow_{2g_g} e_2$. If ? has the value l, the events originate at remote sites and comparison yields $e_1 \parallel_{2g_g} e_2$, because the global representatives are less than two clock ticks apart.

The following list summarises the assumptions for distributed systems:

- Each site has a single local clock. The local clocks are synchronised with precision Π.
- Events are partially ordered; that is, two remote events may be ordered (distance more than two global clock ticks) or concurrent (distance less than two global clock ticks).

$2g_g$-restricted temporal order defines a partial ordering on events based on *physical clocks*, whereas Lamport's causal order (see Section 3.1.2) defines a partial ordering on events based on *logical clocks*.

6.3 Timestamps in Distributed Systems

As discussed in the previous chapter, each event is associated with a timestamp indicating when and where it occurred. The detection of composite events is based on timestamps, that means, timestamps are used to establish the temporal relation between primitive and/or composite events originating at the same or at different sites. When a composite event is detected, its timestamp is derived from the timestamps of constituent events. However, $2g_g$-restricted temporal order and concurrency, $\rightarrow_{2g_g}$ and $\parallel_{2g_g}$, are used to establish the temporal relation between primitive events originating at the same or at different sites, but not between composite events. Therefore, since composite events are composed of primitive and/or other composite events and event operators, $\rightarrow_{2g_g}$ and $\parallel_{2g_g}$ have to be extended in order to deal with both primitive and composite events. The semantics of the event operators sequence (;) and concurrency ($\parallel$) depends on these extended versions of $\rightarrow_{2g_g}$ and $\parallel_{2g_g}$.

In order to examine the semantics of the event operators, it is necessary to first define the structure of timestamps for primitive and composite events. Second, the

Seq	*Con*	*Description*
;	$\|\|$	event operators
$\rightarrow$	$\|\|$	order relations between primitive events, based on causal order
$\rightarrow_{2g_g}$	$\|\|_{2g_g}$	order relations between primitive events, based on temporal order ($2g_g$-precedence)
$<$	$\sim$	temporal relations between timestamps for primitive and composite events

Table 6.1: Overview of Sequence and Concurrency Operators

temporal relationships between timestamps for primitive and composite events are defined in terms of two relations: *happened-before* $< \subseteq T \times T$ and *concurrency* $\sim \subseteq T \times T$, where T denotes the set of timestamps. $<$ represents the extension of $\rightarrow_{2g_g}$ and $\sim$ represents the extension of $\|_{2g_g}$. Finally, the derivation of "complex" timestamps for particular composite events involving conjunction (,), disjunction (|), or concurrency (||) event operators is discussed: the *joining procedure.* Table 6.1 records the different operators for sequence and concurrency.

6.3.1 Requirements for Timestamps

The following list of requirements for timestamps can be deduced from the previous considerations:

1. *Timestamps must contain information on their site(s) of origin.* Comparing timestamps allocated at a single site and timestamps allocated at remote sites may yield different results regarding the ordering of events. Two events whose timestamps are less than $2g_g$ apart are ordered, if they occurred at a single site, and concurrent, if they occurred at remote sites.

2. *Timestamps must contain local and global representatives.* The local representative of a timestamp is not meaningful in a global context, that is, if the timestamp is to be compared with a remote timestamp. However, it is needed for comparing local timestamps.

3. *Timestamps of global composite events may consist of several "primitive" timestamps.* The timestamp of a global event is determined by the latest timestamp

participating in its occurrence. However, in a distributed system there may be no single event determining the latest timestamp; events at multiple sites may happen virtually "at the same time", contributing to the occurrence of a global event and hence, participating in the timestamp.

Local clock ticks and local times (see Section 6.1) are meaningful in a local context only, since the granularity of local clocks is smaller than the precision Π. Therefore, local times have to be transformed into global times in order to be comprehensible by remote sites.

6.3.2 Structure of Timestamps

Definition 6.3.1 (Local clock tick) A *local clock tick* lt_k of a local clock k represents a moment in time, reckoned as a number of clock ticks of granularity g_k since some epoch or starting point.

Definition 6.3.2 (Global time) The global clock granularity g_g is given. The *global time* gt_k of a local clock tick lt_k is the local clock tick expressed according to the standard (Gregorian) calendar with respect to some time zone (e.g. UTC, Universal Time Coordinated) and truncated to granularity g_g.

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

Instead of being truncated to the next lowest global clock tick, the local time could also be rounded to the nearest global clock tick. However, truncation or rounding has to be done consistently throughout the system.

A global time of granularity $g_g = 1/100sec$ could, for example, be represented as day/month/year,hours:minutes:seconds.hundredth. If the local clock granularity is in the order of microseconds, the global time is derived by cutting off the last digit of the fraction. Henceforth, a global clock granularity of $g_g = 1/100sec$ is applied in the examples.

After introducing the different notions of time, the definition of a timestamp is given next.

Definition 6.3.3 (Timestamp) The set of sites S is given. A *timestamp* $T(e)$ of an event e is a partial function $T(e) : S \to (global, local)$ defining a local clock tick $local_s$ and its corresponding global time $global_s$ with $global_s = gt_s(local_s)$ for each site s participating in the timestamp. The distance between the minimum and the maximum global time is not more than one global clock tick.

Notation The *base* and the *limit* of a timestamp $T(e)$ are global times defined as follows:

$$base(T(e)) = MIN\{T(e)(s).global \mid s \in S\}$$

$$limit(T(e)) = MAX\{T(e)(s).global \mid s \in S\}$$

The domain of a timestamp $T(e)$ is called the *(site) set.* For each s in the site set, $T(e)(s).local$ determines the local clock tick and $T(e)(s).global$ the corresponding global time. *base* and *limit* are derived values and denote relevant global times, namely the minimum and the maximum global time. The following condition is valid for all timestamps:

$$limit(T(e)) - base(T(e)) \leq 1$$

Definition 6.3.4 (Timestamp of a primitive event) Given a primitive event e occurring on site s at local clock tick lt_s, the *timestamp of the primitive event* $T(e)$ is determined as follows[1]:

- $T(e)(s).local = lt_s$ and $T(e)(s).global = gt_s(lt_s)$
- $T(e)(s').local = \perp$ and $T(e)(s').global = \perp$ for all $s' \neq s$.

The timestamp of a primitive event is represented as $T(e) = (s, (gt_s(lt_s), lt_s))$.

Example 6.3.5 The timestamp of a primitive event e originating at site k may look as follows:

$$T(e) = (k, (19/10/95, 2:32:27.32, 23991548127))$$

6.3.3 Temporal Relationship between Timestamps

The temporal relationship between timestamps of primitive events can be determined directly from Definitions 6.2.2 and 6.2.3. If the timestamps of two primitive events $T(e_1)$ and $T(e_2)$ originate at the same site k, their local clock ticks $T(e_1)(k).local$ and $T(e_2)(k).local$ are compared in order to determine $T(e_1) < T(e_2)$ or $T(e_2) < T(e_1)$. If $T(e_1)$ and $T(e_2)$ originate at remote sites k and l, their global times $T(e_1)(k).global$ and $T(e_2)(l).global$ are explored. A distance of two or more global clock ticks implies $<$, a distance of less than two global clock ticks implies $\sim$. However, timestamps of global events may have a more complex structure, consisting of several "primitive" timestamps. Hereafter, such timestamps are called

[1] $\perp$ means *undefined*

non-atomic, as opposed to *atomic* timestamps which correspond to primitive events. The constituent "primitive" timestamps of a non-atomic timestamp are called the *component timestamps*.

Definition 6.3.6 (Component timestamp) A *component timestamp* t of a timestamp T(e) is a triple (*site, global, local*), where $site \in domain(T(e))$, $global = T(e)(site).global$ and $local = T(e)(site).local$.

Before the temporal relationship between non-atomic timestamps is discussed in detail, it is worthwhile to stress that there are physical limitations to the ordering of events in distributed systems. Two observers at two different sites may witness different event patterns. In distributed systems with synchronised local clocks, the inaccuracy originates from multiple events happening within a $2g_g$-interval. The objective of the forthcoming definitions is to extend Definitions 6.2.2 and 6.2.3 in a suitable way, in order to provide a consistent and sound view on temporal relationships between timestamps for all observers. The following requirements apply to the concurrency-relation ($\sim$) between timestamps:

- if $T(e_1)$ and $T(e_2)$ are atomic, concurrency is defined in accordance with Definition 6.2.2
- no two component timestamps $t_1 \in T(e_1)$ and $t_2 \in T(e_2)$ are two or more global clock ticks apart
- any two component timestamps $t_1 \in T(e_1)$ and $t_2 \in T(e_2)$ originating at the same site denote the same event

The requirements for concurrency are reasonably straightforward. First, all component timestamps must lie within a $2g_g$ time interval. Second, all distinct component timestamps must originate at distinct sites. The following requirements apply to the happened-before-relation ($<$) between timestamps:

- if $T(e_1)$ and $T(e_2)$ are atomic, happened-before is defined in accordance with Definition 6.2.2
- there are at least two component timestamps $t_1 \in T(e_1)$ and $t_2 \in T(e_2)$

  i originating at the same site and t_1's global time is at least one global clock tick smaller than t_2's global time *ii* originating at different sites and t_1's global time is at least two global clock ticks smaller than t_2's global time

- there are no two component timestamps $t_1 \in T(e_1)$ and $t_2 \in T(e_2)$

n_i originating at the same site and t_2's local clock tick is at least one local clock tick smaller than t_1's local clock tick

n_ii originating at different sites and t_2's global time is at least one global clock tick smaller than t_1's global time

Requirements *i* and *ii* assure that there is at least one pair of component timestamps that implies sequentiality. *i* considers timestamps originating at the same and *ii* considers timestamps originating at different sites. All other component timestamps may lie within a $2g_g$ time interval. One may ask why it is not sufficient to define *i* as "...t_1's local clock tick is at least one local clock tick smaller than t_2's local clock tick". An example is given next. Requirements *n_i* and *n_ii* assure that there is no pair of component timestamps that implies a reverse ordering. The second and third examples justify the definition of *n_i* and *n_ii* respectively. It should be noted, that requirements *i*, *n_i*, and *n_ii* are only relevant for timestamps whose *base*-values are zero or one global clock ticks apart. All other cases are covered by *ii*.

Example 6.3.7 Consider three timestamps $T(e_1)$, $T(e_2)$ and $T(e_3)$, where the values relating to "..." are equivalent in all component timestamps.

Timestamp	*Site*	*Global Time*	*Local Clock Tick*
$T(e_1)$	k	...	...1
	m	...	...2
$T(e_2)$	l	...	...1
	k	...	...2
$T(e_3)$	m	...	...1
	l	...	...2

If *i* corresponds to "...t_1's local clock tick is at least one local clock tick smaller than t_2's local clock tick", then $T(e_1)(k) \rightarrow_{2g_g} T(e_2)(k)$ implies $T(e_1) < T(e_2)$, $T(e_2)(l) \rightarrow_{2g_g} T(e_3)(l)$ implies $T(e_2) < T(e_3)$, and $T(e_3)(m) \rightarrow_{2g_g} T(e_1)(m)$ implies $T(e_3) < T(e_1)$. Hence, $T(e_1) < T(e_2) < T(e_3) < T(e_1)$.

Example 6.3.8 Consider two timestamps $T(e_1)$ and $T(e_2)$.

Timestamp	*Site*	*Global Time*	*Local Clock Tick*
$T(e_1)$	k	$\ldots 42$	
	m	$\ldots 43$	$\ldots 2$
$T(e_2)$	m	$\ldots 43$	$\ldots 1$
	l	$\ldots 44$	

Requirement *n_i* is necessary; although the global times of $T(e_1)(k)$ and $T(e_2)(l)$ are two global clock ticks apart and therefore imply $T(e_1) < T(e_2)$, $T(e_1)(m)$'s and $T(e_2)(m)$'s local clock ticks provide contradictory information.

Example 6.3.9 Consider three timestamps $T(e_1)$, $T(e_2)$ and $T(e_3)$.

Timestamp	*Site*	*Global Time*	*Local Clock Tick*
$T(e_1)$	k	$\ldots 82$	
	m	$\ldots 83$	
$T(e_2)$	l	$\ldots 82$	
	k	$\ldots 83$	
$T(e_3)$	m	$\ldots 82$	
	l	$\ldots 83$	

Without requirement *n_ii*, $T(e_1)(k) \rightarrow_{2g_g} T(e_2)(k)$, $T(e_2)(l) \rightarrow_{2g_g} T(e_3)(l)$, and $T(e_3)(m) \rightarrow_{2g_g} T(e_1)(m)$ imply $T(e_1) < T(e_2) < T(e_3) < T(e_1)$.

The examples above present timestamps which are neither sequential nor concurrent. Hereafter, such timestamps are called *unrelated.* Unrelated timestamps are exceptional cases, which occur when the *base*-values of two (or more) timestamps, at least one of which is non-atomic, are zero or one global clock ticks apart. In one sense such timestamps should be concurrent. However they are not, because they contain distinct component timestamps which originate at the same site and hence imply sequentiality. Considering such cases in the ordering of non-atomic timestamps, as shown in the examples above, would lead to inconsistent results in the form of circular orderings. The following definitions formalise the notions of happened-before ($<$), concurrency ($\sim$), and unrelatedness between timestamps.

Definition 6.3.10 (Temporal relationship) On the basis of the $2g_g$-precedence time model, the *temporal relationship* between two timestamps $T(e_1)$ and $T(e_2)$ is

	reflexivity	*symmetry*	*transitivity*
$<$	no	no	yes
$\sim$	yes	yes	no

Table 6.2: Applicability of Mathematical Rules

defined as follows:

$$\begin{array}{llll}
T(e_1) \sim T(e_2) & \text{iff} & & \forall t_1 \in T(e_1)\ \forall t_2 \in T(e_2), \\
 & & & t_1.\mathit{site} = t_2.\mathit{site} \textbf{ and } t_1.\mathit{local} = t_2.\mathit{local} \textbf{ or} \\
 & & & t_1.\mathit{site} \neq t_2.\mathit{site} \textbf{ and } \mid t_1.\mathit{global} - t_2.\mathit{global} \mid < 2g_g \\
T(e_1) < T(e_2) & \text{iff} & & T(e_1), T(e_2) \text{ atomic } \textbf{and} \\
 & & & T(e_1).\mathit{site} = T(e_2).\mathit{site} \textbf{ and } T(e_1).\mathit{local} < T(e_2).\mathit{local} \\
 & & \textbf{or} & \exists t_1 \in T(e_1)\ \exists t_2 \in T(e_2), \\
 & & & t_1.\mathit{site} = t_2.\mathit{site} \textbf{ and } t_1.\mathit{global} < t_2.\mathit{global} \textbf{ or} \\
 & & & t_1.\mathit{site} \neq t_2.\mathit{site} \textbf{ and } t_1.\mathit{global} < t_2.\mathit{global}\ - 1g_g \\
 & & & \textbf{and } \forall t_1 \in T(e_1)\ \forall t_2 \in T(e_2), \\
 & & & \quad t_1.\mathit{site} = t_2.\mathit{site} \textbf{ and } t_1.\mathit{local} \leq t_2.\mathit{local} \textbf{ or} \\
 & & & \quad t_1.\mathit{site} \neq t_2.\mathit{site} \textbf{ and } t_1.\mathit{global} \leq t_2.\mathit{global}
\end{array}$$

Definition 6.3.11 (Unrelated timestamps) Given two timestamps $T(e_1)$ and $T(e_2)$. If neither $T(e_1) < T(e_2)$ nor $T(e_2) < T(e_1)$ nor $T(e_1) \sim T(e_2)$, then the timestamps are *unrelated*, $T(e_1) \bowtie T(e_2)$.

Table 6.2 summarises the applicability of reflexivity, symmetry, and transitivity for happened-before ($<$) and concurrency ($\sim$).

6.3.4 Joining Procedure for Timestamps

The timestamp of a composite event is determined by the latest timestamp participating in its occurrence. If there are multiple latest timestamps, that is, if the timestamps in question are either concurrent or unrelated, they all contribute to the timestamp of the composite event. The procedures for deriving the resulting *joined timestamp* are illustrated next; first for concurrent timestamps, and second for unrelated timestamps. The joining procedures must respect the structure of timestamps, i.e. the distance between the minimum and the maximum global time of the resulting timestamp should not be more than one global clock tick. Moreover, since a timestamp is a partial function, each site has at most one entry in the resulting timestamp.

Joining Concurrent Timestamps

The following proposition shows that for any two concurrent timestamps the minimum and the maximum global time are not more than one global clock tick apart.

Proposition 6.3.12 If $T(e_1) \sim T(e_2)$, then at most two adjacent global clock ticks cover the timestamps $T(e_1)$ and $T(e_2)$.

Proof The proposition follows directly from the definition of $\sim$; no two component timestamps are two or more global clock ticks apart. $\Box$

Two distinct events at the same site are never concurrent. However, concurrency is reflexive. Therefore, two concurrent timestamps may each contain a component timestamp originating at the same site denoting the same event. The joined timestamp will simply inherit the common component timestamp.

Definition 6.3.13 (Joining concurrent timestamps) Suppose given two timestamps such that $T(e_1) \sim T(e_2)$. The joined timestamp $\hat{T} = T(e_1) \cup T(e_2)$ is defined as follows:

$$joinset := set_1 \cup set_2$$

$$\text{For all } s \in joinset, \quad \hat{T}.(s) := \begin{cases} T(e_1)(s) & \forall s \in set_1 \\ T(e_2)(s) & \text{otherwise} \end{cases}$$

The joining procedure for concurrent timestamps is straightforward; the component timestamps are joined and duplicates are eliminated.

Example 6.3.14 Given two timestamps
$T(e_1) = (k, (19/10/95, 2:32:27.32, 23991548127))$ and
$T(e_2) = (l, (19/10/95, 2:32:27.33, 17233741390))$.
$T(e_1)$ and $T(e_2)$ are concurrent, $T(e_1) \sim T(e_2)$, and the joined timestamp $\hat{T} = T(e_1) \cup T(e_2)$ is

$$\{ (k, (19/10/95, 2:32:27.32, 23991548127)), \\ (l, (19/10/95, 2:32:27.33, 17233741390)) \}$$

Joining Unrelated Timestamps

Unrelated timestamps occur because of a mismatch between component timestamps originating at the same and at different sites. The latter imply the concurrency of the corresponding events, whereas the component timestamps originating at the same site are distinct and are therefore not concurrent but sequential. Unrelated timestamps underline the fact that in a distributed system, observers at different sites may witness different event patterns if the events occur closely together.

It should be noted that unrelated timestamps are not likely to occur frequently; two events occur in quick succession at the same site (i.e. they are less than two global clock ticks apart), they then participate in the timestamps of two different composite events, and these composite events both participate in the detection of a third composite event.

Examples 6.3.7 to 6.3.9 illustrate different cases of unrelated timestamps. In the first example, the minimum and the maximum global time of the events are equal, in the second example, they are two global clock ticks apart and in the third example, they are one global clock tick apart. The following proposition shows that for any two unrelated timestamps the minimum and the maximum global time cannot be more than two global clock ticks apart.

Proposition 6.3.15 If $T(e_1) \bowtie T(e_2)$, then at most three adjacent global clock ticks cover the timestamps $T(e_1)$ and $T(e_2)$.

Proof by contradiction Assume without loss of generality that $base_1 \leq base_2$[2]. If $T(e_1)$ and $T(e_2)$ cover four or more adjacent global clock ticks, then

$$limit_2 - base_1 \geq 3g_g \Longrightarrow base_2 - limit_1 \geq 1g_g$$
$$\text{(since } base \leq limit \leq base + 1g_g \text{ for all timestamps)}$$

Hence, the timestamps must be sequential, $T(e_1) < T(e_2)$, since $base_1$ is more than $2g_g$ smaller than $limit_2$ and there cannot be two component timestamps which are reversely ordered (because of $base_2 > limit_1$). □

There are two complications when trying to join two unrelated timestamps: first, the minimum and maximum global time may be two global clock ticks apart and second, the two timestamps contain different entries for the same site(s). The latter

[2]For abbreviation, the timestamp identification is written as subindex (e.g. $base_1$ instead of $base(T(e_1))$).

point is inherent in unrelated timestamps; component timestamps originating at the same site (may) cause inconsistent, circular orderings. The joining procedure must therefore handle both problems. The idea of the joining procedure for unrelated timestamps is to keep the information that is most relevant to future time. That means, if the minimum and maximum global time are two global clock ticks apart, the component timestamps of the earliest global time are disregarded. Also, if there are two different entries for the same site, the later entry is kept.

Definition 6.3.16 (Joining unrelated timestamps) Suppose given two timestamps such that $T(e_1) \bowtie T(e_2)$. The joined timestamp $\hat{T} = T(e_1) \cup T(e_2)$ is defined as follows:

```
Assume without loss of generality that $base_1 \leq base_2$.
if $limit_2 - base_1 = 2g_g$ then
    Delete all $s \in set_1$ with $T(e_1)(s).global = base_1$
fi
$joinset := set_1 \cup set_2$
```

$$\text{For all } s \in joinset, \quad \hat{T}.(s) := \begin{cases} T(e_1)(s) & \forall s \in set_1 \backslash set_2 \\ T(e_2)(s) & \forall s \in set_2 \backslash set_1 \\ MAX\,\{T(e_1)(s), T(e_2)(s)\} & \forall s \in set_1 \cap set_2 \end{cases}$$

The joining procedure for unrelated timestamps proceeds as follows: in the **if**-clause check whether the minimum and the maximum global time ($base_1$ and $limit_2$) are two global clock ticks apart. If this is the case, all entries for $base_1$ are deleted. The remaining site-entries are joined keeping the maximum when there are two different entries for the same site.

Example 6.3.17 Given are the two timestamps from Example 6.3.8
$T(e_1) = \{(k, (\ldots 42, \quad)), (m, (\ldots 43, \ldots 2))\}$ and
$T(e_2) = \{(m, (\ldots 43, \ldots 1)), (l, (\ldots 44, \quad))\}$.
$T(e_1)$ and $T(e_2)$ are unrelated, $T(e_1) \bowtie T(e_2)$, and the joined timestamp $\hat{T} = T(e_1) \cup T(e_2)$ is

$$\{(m, (\ldots 43, \ldots 2)), (l, (\ldots 44, \quad))\}$$

6.4 Timestamps in Distributed Systems – Simplified Semantic Model

The considerations in the previous section are based on the assumption that two primitive events at the same site can occur within one global clock tick. The reason for this is that the global clock granularity, which is determined by the precision of the synchronised local clocks, is large in comparison with the local clock granularity. Although this assumption is valid, some applications do not demand such a fine granularity. For example, most research projects introduced in Chapter 2 work with a granularity of one second, which is large in comparison with the precision achieved by common clock synchronisation protocols. In these cases, a simplified semantic model for composite events can be applied. This semantics is developed hereafter [**?**].

Assumption There are no two primitive events e_1 and e_2, $e_1 \neq e_2$, occurring at the same site $s \in S$ at the same global time $gt_s\langle e_1\rangle = gt_s\langle e_2\rangle$.

The simplified semantic model can only be applied if the frequency of event occurrences is sufficiently low, that is, if there is less than one event per global clock tick. Consider a distributed debugging system, where there are about 30000 events/minute. If the global clock granularity is set to $g_g = 1/100sec$, there are on average 5 events/g_g. Hence, the simplified semantic model cannot be applied.

In the general semantic model, local clock ticks are employed for establishing the temporal relation between primitive events originating at the same site. However, in the simplified semantic model the temporal relation can be established on the basis of global times, since any two primitive events originating at the same site are assigned different global clock ticks. Therefore, local clock ticks are redundant. This simplifies the structure and handling of timestamps.

6.4.1 Structure of Timestamps

Definition 6.4.1 (Timestamp) The set of sites S is given. A *timestamp* $T(e)$ of an event e is a partial function $T(e) : S \to \mathit{global}$ defining a global time $\mathit{global}_s = gt_s(lt_s)$ for each site s participating in the timestamp. The distance between the minimum and the maximum global time is not more than one global clock tick.

In comparison with Definition 6.3.3, only a global time is defined for each site participating in the timestamp. Since $\mathit{limit}(T(e)) - \mathit{base}(T(e)) \leq 1$, there is a maximum of two different global times within one timestamp, the *base* and the *limit*.

Definition 6.4.2 (Timestamp of a primitive event) Given a primitive event e occurring on site s at global time $gt_s(lt_s)$, the *timestamp of the primitive event* $T(e)$ is determined as follows:

- $T(e)(s).global = gt_s(lt_s)$
- $T(e)(s').global = \perp$ for all $s' \neq s$.

Example 6.4.3 The timestamp of a primitive event e originating at site k may look as follows:

$$T(e) = (k, 19/10/95, 2:32:27.32)$$

6.4.2 Temporal Relationship between Timestamps

Definition 6.4.4 (Component timestamp) A *component timestamp* t of a timestamp $T(e)$ is a tuple (*site, global*), where $site \in domain(T(e))$ and $global = T(e)(site).global$.

The requirements for concurrency ($\sim$) and happened-before ($<$) in the general semantic model (see Section 6.3.3) are of course valid in the simplified semantic model. In the general semantic model two local clock ticks may be compared only if they are associated with the same site: in the simplified semantic model such local clock ticks are represented by the global times at the site. What is true for local clock ticks in the general semantic model becomes true for global times. Hence, the equivalent of Definition 6.3.10 is derived by substituting references to *t.local* by *t.global* and by eliminating redundancies.

Definition 6.4.5 (Temporal relationship) On the basis of the $2g_g$-precedence time model, the *temporal relationship* between two timestamps $T(e_1)$ and $T(e_2)$ is defined as follows:

$$\begin{array}{lll}
T(e_1) \sim T(e_2) & \text{iff} & \forall t_1 \in T(e_1)\ \forall t_2 \in T(e_2), \\
 & & t_1.\mathit{site} = t_2.\mathit{site} \ \textbf{and}\ t_1.\mathit{global} = t_2.\mathit{global}\ \textbf{or} \\
 & & t_1.\mathit{site} \neq t_2.\mathit{site}\ \textbf{and}\ \mid t_1.\mathit{global} - t_2.\mathit{global} \mid < 2g_g \\
T(e_1) < T(e_2) & \text{iff} & \exists t_1 \in T(e_1)\ \exists t_2 \in T(e_2), \\
 & & t_1.\mathit{site} = t_2.\mathit{site}\ \textbf{and}\ t_1.\mathit{global} < t_2.\mathit{global}\ \textbf{or} \\
 & & t_1.\mathit{site} \neq t_2.\mathit{site}\ \textbf{and}\ t_1.\mathit{global} < t_2.\mathit{global}\ - 1g_g \\
 & \textbf{and} & \forall t_1 \in T(e_1)\ \forall t_2 \in T(e_2), \\
 & & t_1.\mathit{global} \leq t_2.\mathit{global}
\end{array}$$

There are unrelated timestamps in the simplified semantic model. In any such case the *base*-values of two (or more) non-atomic timestamps must be equal. Example 6.3.9 shows three timestamps which are unrelated in the simplified semantic model. However, Examples 6.3.7 and 6.3.8 are not, because there the fact that they are unrelated depends on local clock ticks.

6.4.3 Joining Procedure for Timestamps

The joining of concurrent and unrelated timestamps is more intuitive than in the general semantic model because the timestamps cover not more than two adjacent global clock ticks.

Proposition 6.4.6 If neither $T(e_1) < T(e_2)$ nor $T(e_2) < T(e_1)$, then at most two adjacent global clock ticks cover the timestamps $T(e_1)$ and $T(e_2)$.

Proof by contradiction The simplified semantic model is a special case of the general semantic model; two unrelated timestamps may cover no more than three adjacent global clock ticks (see Propositions 6.3.12 and 6.3.15).

Assume without loss of generality that $\mathit{base}_1 \leq \mathit{base}_2$. Suppose that $T(e_1)$ and $T(e_2)$ cover three adjacent global clock ticks, so that $\mathit{limit}_2 - \mathit{base}_1 = 2g_g$. Certainly there are component timestamps $t_1 \in T(e_1)$ and $t_2 \in T(e_2)$ such that $t_1.\mathit{global} = \mathit{base}_1$, $t_2.\mathit{global} = \mathit{limit}_2 > t_1.\mathit{global} + 1g_g$. Further, $\mathit{limit}_1 \leq \mathit{base}_1 + 1g_g = \mathit{limit}_2 - 1g_g \leq \mathit{base}_2$: hence, for all component timestamps $t_1 \in T(e_1)$ and $t_2 \in T(e_2)$

$$t_1.\mathit{global} \leq \mathit{limit}_1 \leq \mathit{base}_2 \leq t_2.\mathit{global}$$

According to Definition 6.4.5 the timestamps must be sequential, $T(e_1) < T(e_2)$. The reason that unrelated timestamps cannot arise in such a case in the simplified semantic model is that there are no local clock ticks which might imply an inverse ordering. $\square$

The following definition shows the joining procedure for concurrent and for unrelated timestamps in the simplified semantic model. Since both concurrent and unrelated timestamps do not cover more than two adjacent global clock ticks, the same joining procedure can be applied.

Definition 6.4.7 (Joined timestamps) Suppose given two timestamps such that neither $T(e_1) < T(e_2)$ nor $T(e_2) < T(e_1)$. The joined timestamp $\hat{T} = T(e_1) \cup T(e_2)$ is defined as follows:

$$\mathit{joinset} := \mathit{set}_1 \cup \mathit{set}_2$$

$$\text{For all } s \in \mathit{joinset}, \quad \hat{T}.(s) \;:=\; \begin{cases} T(e_1)(s) & \forall s \in \mathit{set}_1 \backslash \mathit{set}_2 \\ T(e_2)(s) & \forall s \in \mathit{set}_2 \backslash \mathit{set}_1 \\ \mathit{MAX}\ \{T(e_1)(s), T(e_2)(s)\} & \forall s \in \mathit{set}_1 \cap \mathit{set}_2 \end{cases}$$

6.5 Semantics of Composite Events

The previous sections established the foundations for defining the formal semantics of composite events; the structure and handling of timestamps were investigated in the general and in the simplified semantic model. The following definition formalises the semantics of composite events for both semantic models; that is, the definitions of timestamps, concurrency ($\sim$), and happened-before ($<$) relate either to the general semantic model (see Section 6.3) or to the simplified semantic model (see Section 6.4).

Definition 6.5.1 (Semantics of composite events) On the basis of the $2g_g$-precedence time model, the semantics of composite events is defined as follows. The first part of the right-hand side identifies under which pre-conditions a composite event of that event type is detected and the second part, indicated by $\Rightarrow$, presents the resulting timestamp:

$E_1 \ , \ E_2$	iff	$e_1 \in E_1$ and $e_2 \in E_2$
		$\Rightarrow T(e_2)$ iff $T(e_1) < T(e_2)$
		$\Rightarrow T(e_1)$ iff $T(e_2) < T(e_1)$
		$\Rightarrow T(e_1) \cup T(e_2)$ otherwise
$E_1 \mid E_2$	iff	$e_1 \in E_1$ or $e_2 \in E_2$
		[or both, if $T(e_1) \sim T(e_2)$ or $T(e_1) \bowtie T(e_2)$]
		$\Rightarrow T(e_1)$ or $T(e_2)$
		[or $T(e_1) \cup T(e_2)$]
$E_1 \ ; \ E_2$	iff	$e_1 \in E_1$ and $e_2 \in E_2$ and $T(e_1) < T(e_2)$
		$\Rightarrow T(e_2)$
$E_1 \parallel E_2$	iff	$e_1 \in E_1$ and $e_2 \in E_2$ and $T(e_1) \sim T(e_2)$
		$\Rightarrow T(e_1) \cup T(e_2)$
$E_1 \ ^\star \ E_2$	iff	$e_1^i \in E_1 \ (i \geq 0)$ and $e_2 \in E_2$ and $T(e_1^i) < T(e_2)$
		$\Rightarrow T(e_2)$
$E_1 \ ^+ \ E_2$	iff	$e_1^i \in E_1 \ (i \geq 1)$ and $e_2 \in E_2$ and $T(e_1^i) < T(e_2)$
		$\Rightarrow T(e_2)$
E_1 ; NOT E_2 ; E_3	iff	$e_1 \in E_1$ and $e_3 \in E_3$ and $T(e_1) < T(e_3)$ and
		no $e_2 \in E_2$ with $T(e_1) < T(e_2) < T(e_3)$
		$\Rightarrow T(e_3)$

Conjunction $\mathbf{E_1 \ , \ E_2}$: A conjunction demands both events, $e_1 \in E_1$ and $e_2 \in E_2$, to happen regardless of their order. The timestamp of a conjunction is the later timestamp of e_1 and e_2, if the temporal order of e_1 and e_2 can be decided. Otherwise the two timestamps are joined.

Disjunction $\mathbf{E_1 \mid E_2}$: There are two possible semantics for a disjunction: *exclusive-or* or *inclusive-or* (in square brackets). An exclusive-or is detected when either $e_1 \in E_1$ or $e_2 \in E_2$ occurs. The timestamp of that occurrence becomes the timestamp of the disjunction. In addition, an inclusive-or demands that both events are considered, that is, their timestamps are joined, if $e_1 \in E_1$ and $e_2 \in E_2$ both occur and their temporal order cannot be established. Which of the two semantics is realised in a specific distributed system depends on application-demands.

Sequence $\mathbf{E_1 \ ; \ E_2}$: Inherently, a sequence between events represents the happened-before relation ($<$) between timestamps. Event $e_1 \in E_1$ happens before event $e_2 \in E_2$, if e_1's timestamp is smaller than e_2's timestamp. Since e_2's timestamp is the later timestamp, it becomes the timestamp of $E_1 \ ; \ E_2$.

Concurrency $\mathbf{E_1 \parallel E_2}$: A concurrency between events is equivalent to a concurrency ($\sim$) between timestamps. No temporal order between $e_1 \in E_1$ and $e_2 \in E_2$ can be inferred from their timestamps. Therefore, both e_1 and e_2 are latest events and their timestamps are joined.

Iteration $\mathbf{E_1 \ ^\star \ E_2}$ and $\mathbf{E_1 \ ^+ \ E_2}$: An iteration is a special case of a sequence, in which $e_2 \in E_2$ serves as a delimitation following a number of occurrences of $e_1 \in E_1$ (zero or more in the case of * and one or more in the case of $^+$). Consequently, the timestamp is that of e_2.

Negation $\mathbf{E_1 \ ; \ NOT \ E_2 \ ; \ E_3}$: As above, a negation is a special case of a sequence. Therefore, $e_3 \in E_3$ defines the timestamp of the negation.

6.6 Summary

This chapter addressed the semantics of events in distributed systems. Since the semantics is given in terms of timestamps concerning physical time, the notions of physical time and temporal order in distributed systems were examined. The $2g_g$-precedence time model was presented, which makes it possible to approximate a global time base. The structure and handling of timestamps were illustrated next, first in the general semantic model, and second in the simplified semantic model. The latter is applicable in specific distributed systems, where the frequency of event occurrences is lower than the precision of the local clocks; that is, there is less than one event within one global clock tick. Finally, the semantics of composite events was formally defined. The semantics is valid for both the general and the simplified semantic model.

Chapter 7

Detection of Events

Event detection is the process of recognising the occurrence of an event and collecting and recording its parameters including a timestamp [?]. In the context of this thesis, timestamps record when and where an event occurred. The recognition of primitive event occurrences was discussed in Section 2.1. The recognition of composite event occurrences is an incremental process, based on some underlying computational model. For each composite event, there is a specific event detector reflecting its structure. Each time a relevant event occurrence is signalled, it is checked whether the event detector can progress. A certain final state indicates the occurrence of an event. On the recognition of an event, the system reacts accordingly: in active database systems, the recognised event is passed to the rule manager for evaluating the condition and, if it holds, triggering the action of an ECA rule; in distributed debugging systems, the system may be halted and debugging information is displayed to the user.

7.1 Goals of Event Detection in Distributed Systems

The following list summarises the goals of event detection in distributed systems:

1. *Event detection should run in the background of other system functions, causing as little overhead as necessary.* Monitoring system behaviour is a prerequisite for the smooth implementation of a number of system functions, such as integrity/security enforcement, constraint management, and the realisation of reactive behaviour in general (in active database systems) or program debugging, testing, performance management, and configuration management (in distributed monitoring systems). However, monitoring is not visible to the

user and should therefore not interfere with his/her actions. Two main possibilities for reducing the overhead of event monitoring are *garbage collection of obsolete events* and *minimisation of the event traffic.*

2. *Event detection should be distributed.* Distributing event detectors increases the autonomy of sites, improves system performance, that is, response times, improves the reliability, availability and robustness, and finally, supports system evolution. Hence, event detectors should be distributed.

3. *Event detection should respect the special characteristics of distributed systems.* Distributed systems are characterised by a lack of global time, autonomy of sites, message delays between sites, and independent failure modes [Bac92]. This implies that events in a distributed system are partially ordered, that is, events may occur concurrently at different sites. Moreover, observed events from different sites experience different message delays, that is, the arrival order at event detectors does not reflect the occurrence order. In other words, event detection in distributed systems is inherently different from event detection in centralised systems and should therefore be considered separately.

4. *Event detection algorithms should reflect user and application needs.* There are different ways for handling the special characteristics of distributed systems: local clocks may or may not be synchronised, messages from failed or delayed sites may or may not be considered, etc. Hence, different event detection policies can be identified, depending on user and application needs.

7.2 Architecture

Before the detection algorithms are presented, the model of a global event detection system is illustrated in Figure **??**. There is a distributed system of n sites, each site containing a *local event detector (LED)* and a *global event detector (GED)*. Local event detectors detect local events and consequently employ "simple" centralised detection mechanisms. Detected local events are *notified* at local and/or remote global event detectors. The global event detectors must have *registered* their interest in a specific event type [**?**]. Each global event detector evaluates events received from multiple sites. Detected global events are either signalled to the site's rule manager and/or are sent to registered global event detectors for further evaluation. Hence, the detection of a global event may be distributed to different sites, each site detecting part of the global event.

In this way, the distinctive features of centralised and distributed systems can be isolated from each other. Existing centralised event detectors can be utilised for efficient detection of local events, whereas the global event detectors focus on the detection of "real" global events, and hence deal with the special characteristics of distributed systems. The main focus of this chapter is the development of algorithms for the detection of global composite events, that is, the *GED*-units and their interconnections are investigated. First, a short review of existing centralised event detection mechanisms is given.

7.3 Local Event Detection

Chapter 2 discussed events in active database systems, where event specification and event detection have been considered in centralised systems only. All event detection mechanisms introduced in connection with the different research prototypes are applicable for local event detection and may be used for the implementation of *LED*-units.

7.3.1 Pre-conditions of Local Event Detection

The following points are specific for composite event detection in centralised systems:

- Timestamps are relatively accurate. Although the relation between a timestamp and real physical time is blurred, the relation between different timestamps gives a correct picture of the order in which events happened.
- The arrival order of events at a local event detector corresponds to the occurrence order of events.

7.3.2 Options for Local Event Detection

All research prototypes in Chapter 2 respect the pre-conditions stated in the previous section. Two research prototypes, HiPAC and Ode, use finite state automata for detecting composite events. Primitive events are fed into a finite state automaton one at a time, in the order in which they were signalled at the event detector. SAMOS uses Petri Nets for detecting composite events. Although Petri Nets allow modelling complex concurrent systems, in SAMOS they are used as purely sequential machines evaluating primitive events one after the other, in the order in which they were signalled at the event detector. Finally, Sentinel's event trees are also evaluated strictly sequentially. Although SAMOS and Sentinel associate timestamps denoting the time of occurrence with detected events, the timestamps are not necessary for the detection of composite events. Instead, they are used in the condition and/or the action of an ECA rule.

7.4 Global Event Detection

There is a distributed system of n sites and a set of global event expressions. The global event expressions are distributed to the n global event detectors; one global event expression is assigned to one global event detector, that is, global event expressions are neither split nor replicated. A site containing the global event detector of one specific global event expression is called the *observer site.* Note, that the constituent operands of a global event expression may be either local or global event expressions and hence the detection of an overall global event may take place at multiple sites.

7.4.1 Pre-conditions of Global Event Detection

Composite event detection in distributed systems faces different pre-conditions than composite event detection in centralised systems. The following points are specific for composite event detection in distributed systems:

- Timestamps are inaccurate due to the lack of a global time base. This implies that events in a distributed system are partially ordered, that is, events may occur concurrently.
- The arrival order of events at a global event detector does not correspond to the occurrence order of events. Events arrive from different sites, that is, via different logical channels. Therefore, events experience different message delays. Moreover, if there is one port receiving events from one logical channel, events may arrive concurrently via different *input ports.*
- Events may get lost. If a site fails after an event has occurred but before it is signalled to registered global event detectors, it may not be possible to recover the event. Moreover, events may get lost during transmission.

The first pre-condition was addressed in the previous chapter, where the structure and handling of timestamps were examined and the semantics of event operators explored, considering distributed systems with synchronised local clocks. Finding techniques for dealing with the second pre-condition is the subject of this and the next chapter. The influence of the third pre-condition can be minimised by applying an appropriate communications protocol, such as RPC [Bac92].

7.4.2 Options for Global Event Detection

In active database systems and in distributed debugging systems, there are several evaluation paradigms. The following paradigms have been investigated with respect to their suitability for global event detection:

Finite State Automata (as in HiPAC (Section 2.2), Ode (Section 2.3), and EBBA (Section 3.2)) Finite state automata are not suitable for global event detection. First, finite state automata are efficient recognisers for sequential behaviour, but they cannot represent concurrent events. Second, a finite state automaton is a sequential machine that evaluates events in the order of their occurrence. Hence, all relevant events need to be reported and sorted before entering the automaton. Late events require a rollback of event detection.

Predecessor Automata (as in Data Path Debugging (Section 3.4)) Predecessor automata are similar to, but extend the concept of finite state automata. Although predecessor automata can represent concurrent events, they are not suitable for global event detection. As finite state automata they are sequential machines, that is, all relevant events need to be reported and sorted before entering the automaton. Moreover, predecessor automata can become quite complex [**?**].

Petri Nets (as in SAMOS (Section 2.4)) Petri Nets can represent sequential and concurrent behaviour. Moreover, they can be evaluated concurrently by different threads. Hence, Petri Nets are suitable for global event detection. However, although Petri Nets are ideal for the modelling and the analysis of concurrent systems, they are complex. This has two drawbacks; first it restricts system evolution, and second it is comparatively inefficient for recognising events at runtime [**?**].

Extended Trees (as in Sentinel (Section 2.5), Global Breakpoints (Section 3.3)) A tree is a data structure and no computational model. Therefore, comparing automata and petri nets with trees is not appropriate. However, trees can serve as the basis for event detection, the evaluation semantics being part of application programs. Extended trees can model sequential and concurrent behaviour; different branches of a tree can be evaluated concurrently. Therefore, extended trees are suitable for global event detection.

Initially, Petri Nets were considered as the underlying paradigm for the implementation of global event detectors. This idea was later dismissed. [**?**, **?**] illustrate how (Coloured) Petri Nets are used for the implementation of local event detectors. The developed SAMOS Petri Nets model a purely sequential behaviour. The SAMOS-approach relies strongly on the pre-condition that the arrival order of events at the detector corresponds to the occurrence order of events. As a result, the detection of local composite events as well as the garbage collection of obsolete events are incorporated straightforwardly into the modelled Petri Nets. The different preconditions of global event detection thwarted the application of this solution. Neither the semantics of global event detection nor garbage collection could be modelled straightforwardly with Petri Nets. Hence, it was decided to base the implementation of global event detectors on the evaluation of *trees*. Trees provide a simple structure and allow an efficient implementation. However, the dynamics of event monitoring cannot be expressed within the model itself.

7.4.3 Basic Detection Mechanisms

The detection of global composite events is based on the evaluation of trees. Each global event expression E is transformed into a *global event tree* $GT(E)$, corresponding to its syntactic structure. Nodes are labelled with event operators and leaves are labelled with event expressions. The outermost event operator of a global event expression corresponds to the root node of the global event tree. Its operands, which are event expressions, correspond to the subtrees having their roots in the children of the root node. Again, the outermost event operator of such an operand corresponds to the root note of the subtree and so on. This structure continues recursively until the operands denote indivisible event expressions, in which case corresponding leaves are inserted into the global event tree. An indivisible event expression is either a local or a global event expression. Each node of a global event tree has separate storage for keeping a list of detected sub-events for each of its children; negation-nodes have three children, all other nodes have two children. Leaves have no storage. If a subtree of a global event tree corresponds to a global event expression with parameter restrictions, a *guard* is attached to the corresponding event operator nodes, stating the event parameters whose equality is to be imposed.

At system runtime, occurrences of local event types are signalled from corresponding local event detectors and occurrences of global event types are signalled from corresponding global event detectors. The events may arrive concurrently via different input ports. Newly arrived event occurrences are then injected into the leaves corresponding to their event type and flow upwards in the global event tree. Since the leaves do not have storage, event occurrences are passed on directly to their parents.

Example 7.4.1 Figure **??** shows a snapshot during evaluation of the global event tree of $((LE_1, GE_1); LE_2) \mid (LE_3; \text{NOT } GE_2; LE_4)$, indicating the lists of stored sub-events as linked boxes. Empty lists are represented as empty boxes. The leaves GE_1 and GE_2 refer to global event trees stored and evaluated elsewhere. The other leaves refer to local events.

Algorithm 7.4.2 The evaluation of global event trees at a global event detector site proceeds as follows:

1. An event occurrence is signalled at a port of the global event detector.
2. The event occurrence is inserted into all leaves corresponding to its event type.

For each inserted event occurrence

3. The event occurrence is propagated to the parent node.
4. Wait while the parent node is locked.
5. Lock the parent node and evaluate it, taking into account:
   (a) the event operator [and the guard]
   (b) the inserted event occurrence
   (c) the event occurrences stored in the child lists
   (d) an *evaluation policy*

   If no event is detected at the parent node, continue at 6).
   If an event is detected at the parent node, continue at 7).

6. Store the inserted event occurrence in the corresponding child list and unlock the evaluated node. The algorithm stops.

7. Derive the new event occurrence and unlock the evaluated node.
   If the evaluated node has a parent node, continue at 3).
   If the evaluated node has no parent node, a global composite event is detected and the algorithm stops.

Algorithm 7.4.2 presents the steps involved in the evaluation of the global event trees stored at a global event detector site. At runtime, primitive events occur and are evaluated at their local event detectors. If occurrences of local event types are detected, the local event detectors initiate messages containing the event type, the timestamp and other event parameters, and send these messages to those local or remote global event detectors which have registered an interest in the particular event type. Received event occurrences are evaluated by global event detectors, that is, they (i.e. their timestamp and other event parameters) are inserted into all leaves corresponding to the event type. Leaves serve as entry points and propagate the event occurrences upwards to their parent nodes, where they get evaluated. Evaluation at a node takes into account the stored sub-events as well as the newly arrived event occurrence. The evaluation procedure depends on an *evaluation policy* which is addressed subsequently. If an event is detected, it is again propagated to the parent node and the procedure recurses or, if there is no parent node, it is signalled to the rule manager. The latter case means that an occurrence of a specified global event has been detected. If no event is detected, the event occurrence is inserted into the corresponding list of sub-events. Since global event trees are evaluated concurrently by different threads corresponding to the different input ports of a global event detector, nodes are locked while being evaluated.

An *evaluation policy* determines three different things: *how* to evaluate a node, *when* to evaluate a node, and *which* event occurrences to consider. The first aspect was addressed in the previous chapter, where the semantics of event operators was explored considering a distributed system with synchronised local clocks. The third aspect relates to *event consumption* (see Section 2.1) and is discussed in the following section. Finally the second aspect, *when* to evaluate a node, is concerned with the handling of message delays. It is the subject of the next chapter.

7.4.4 Event Consumption

At the evaluation of an event operator node (step 5 in Algorithm 7.4.2), there may be multiple event occurrences in a corresponding child list. A newly arriving event occurrence may be composable with a number of those event occurrences. In

Example 7.4.1, an event occurrence ge_{11} is composable with both, le_{11} and le_{12}. In order to avoid ambiguities, the event detector must know how to combine event occurrences when evaluating an event operator node. Moreover, the user must be able to choose between different semantics, depending on the application. This can be done by introducing a *parameter context* (see Section 2.1). [**?**, **?**] identify four possibilities :

Chronicle In this context, the oldest available event occurrences are used, that is, event occurrences are combined in chronological order. On the detection of a composite event, the event occurrences used are deleted.

Recent In this context, only the most recent event occurrences are used. On the detection of a composite event, the event occurrences used are deleted. As newer event occurrences of an event type are understood as refinements of older values, "out-of-date" event occurrences are overwritten. Hence, not all occurrences of an event may be used for event detection.

Continuous In this context, all possible combinations of event occurrences are used. Specifically, an event occurrence may contribute to the detection of a composite event more than once. Event occurrences are only deleted, when the corresponding rule is deactivated. Another possibility is that there is a sliding time point, identifying the earliest event occurrences to be considered. All earlier event occurrences are deleted.

Cumulative In this context, all event occurrences are accumulated until a composite event is detected. On the detection of a composite event, the event occurrences used are deleted.

The algorithms in the subsequent chapter present the evaluation of global event trees in the chronicle context only. The reason for choosing the chronicle context is the belief that in most applications the chronological order of event occurrences is of importance and that the correspondence between event occurrences needs to be maintained. This decision is emphasised by the fact that most related work (studied in Chapters 2 and 3) considers this context. However, the other parameter contexts can be realised as in [**?**, **?**].

7.5 Summary

This chapter illustrated the basic detection mechanism for composite events in distributed systems. Local event detectors detect local composite events and global event detectors detect global composite events. The main algorithm for the evaluation of global event trees representing global composite events was presented and the event consumption of event occurrences at event operator nodes was addressed.

Chapter 8

Evaluation Policies: Asynchronous and Synchronous

The arrival order of events at observer sites does not correspond to the occurrence order of events. The changing delays experienced during event transmission are due to the composition of the network, disruptions of the network (network partitioning or network congestion), and disruptions of the sites where events occur (site failures). There are two extreme possibilities for dealing with delayed events:

- ignoring the fact that there may be delayed events and evaluating global event trees as soon as suitable events arrive at an observer site.
- waiting for delayed events and evaluating global event trees only if all relevant events have arrived at an observer site.

The first possibility motivates *asynchronous evaluation* and the second possibility motivates *synchronous evaluation.*

8.1 Asynchronous Evaluation

Definition 8.1.1 (Asynchronous evaluation) A global event tree is evaluated *asynchronously*, if each node is evaluated instantly on the arrival of an event occurrence from a child node.

Events affected by a failure (site failure, network partitioning or network congestion) are delayed until the failure is repaired. Asynchronous evaluation means that nodes are evaluated irrespective of failures. When events arrive at a node, there may be other events with smaller timestamps which have not yet arrived. The node is, however, evaluated instantly. This implies that events from specific child nodes are not necessarily evaluated in the order of their occurrence, that is, in the chronicle parameter context. More recent events with larger timestamps from other child nodes will be handled as soon as they become available. What is done with the delayed events is another matter for decision. They may either be accepted for event detection as soon as they arrive or they may be disregarded.

The main advantage of asynchronous evaluation is that global event trees are evaluated and composite events are detected regardless of remote failures. Delayed events do not cause temporary blocking of the detection procedure. The simplest example is that of a disjunction $E_1 \mid E_2$. If E_2's site has failed, the disjunction can still be detected whenever E_1 is signalled. Therefore, asynchronous evaluation is characterised by immediate consumption, non-blocking detection and good response times. The main disadvantage of asynchronous evaluation is that it does not guarantee event detection in the chronicle or any other parameter context. Whether this can be accepted or not depends on the specific global composite event and the application domain.

8.1.1 Evaluation of Nodes

The evaluation algorithms for event operator nodes are presented next. Due to the special characteristics of asynchronous evaluation, the garbage collection of obsolete events is not incorporated into the evaluation algorithms, but is performed separately. The garbage collection algorithms are presented in Section 8.1.2.

The asynchronous evaluation of disjunction(inclusive-or)-nodes does not seem to be sensible. Since a node is evaluated as soon as suitable events arrive, the semantics is that of a disjunction(exclusive-or). The evaluation of conjunction(,)-, sequence(;)-, concurrency(‖)-, negation(NOT)-, and iteration($^+$)-nodes is similar. The similarities are incorporated into an evaluation template. Operator-specific parts are stated thereafter.

Algorithm 8.1.2 The following procedures present the asynchronous evaluation of event operator nodes:

```
Event_Op  LOCK the node
             ...BLOCK 1 ...
             IF $e_1$ is signalled
```

```
            IF $E_2$'s list is not empty
                ...BLOCK 2 ...
            ELSE
                Insert $e_1$ into $E_1$'s list;
            END
        END
        IF $e_2$ is signalled
            IF $E_1$'s list is not empty
                ...BLOCK 3 ...
            ELSE
                Insert $e_2$ into $E_2$'s list;
            END
        END
    UNLOCK the node
```

```
$E_1$ , $E_2$       ...BLOCK 2 ...
                    $e_2$ = head of $E_2$'s list; delete head of $E_2$'s list;
                    Propagate $e_1$ , $e_2$ to parent node;
                ...BLOCK 3 ...
                    $e_1$ = head of $E_1$'s list; delete head of $E_1$'s list;
                    Propagate $e_1$ , $e_2$ to parent node;
$E_1$ ; $E_2$       ...BLOCK 2 ...
                and there is $e_2$ with $T(e_1) < T(e_2)$
                    Delete $e_2$ from $E_2$'s list;
                    Propagate $e_1$ ; $e_2$ to parent node;
                ...BLOCK 3 ...
                and there is $e_1$ with $T(e_1) < T(e_2)$
                    Delete $e_1$ from $E_1$'s list;
                    Propagate $e_1$ ; $e_2$ to parent node;
$E_1 \| E_2$        ...BLOCK 2 ...
                and there is $e_2$ with $T(e_1) \sim T(e_2)$
                    Delete $e_2$ from $E_2$'s list;
                    Propagate $e_1 \| e_2$ to parent node;
                ...BLOCK 3 ...
                and there is $e_1$ with $T(e_1) \sim T(e_2)$
                    Delete $e_1$ from $E_1$'s list;
                    Propagate $e_1 \| e_2$ to parent node;
$E_1$ ; NOT $E_3$ ; $E_2$
                ...BLOCK 1 ...
                IF $e_3$ is signalled
```

```
                    Insert $e_3$ into $E_3$'s list;
                END
            ...BLOCK 2 ...
                and there is $e_2$ with $T(e_1) < T(e_2)$
                    Delete $e_2$ from $E_2$'s list;
                    IF there is no $e_3$ in $E_3$'s list with $T(e_1) < T(e_3)$ and $T(e_3) < T(e_2)$
                        Propagate $e_1$ ; NOT $e_3$ ; $e_2$ to parent node;
                    END
            ...BLOCK 3 ...
                and there is $e_1$ with $T(e_1) < T(e_2)$
                    Delete $e_1$ from $E_1$'s list;
                    IF there is no $e_3$ in $E_3$'s list with $T(e_1) < T(e_3)$ and $T(e_3) < T(e_2)$
                        Propagate $e_1$ ; NOT $e_3$ ; $e_2$ to parent node;
                    END
$E_1 \ ^+ \ E_2$    ...BLOCK 2 ...
                and there is $e_2$ with $T(e_1) < T(e_2)$
                    Delete $e_2$ from $E_2$'s list;
                    Propagate $e_1 \ ^+ \ e_2$ to parent node;
            ...BLOCK 3 ...
                and there is $e_1$ with $T(e_1) < T(e_2)$
                    Delete all $e_1$'s with $T(e_1) < T(e_2)$ from $E_1$'s list;
                    Propagate $e_1 \ ^+ \ e_2$ to parent node;
```

```
$E_1 \mid E_2$    FOR any event $e \in \{e_1, e_2\}$ signalled
                Propagate $e$ to parent node;
            END;

$E_1 \ ^\star \ E_2$    LOCK the node
                IF $e_1$ is signalled
                    Insert $e_1$ into $E_1$'s list;
                END
                IF $e_2$ is signalled
                    Delete all $e_1$'s with $T(e_1) < T(e_2)$ from $E_1$'s list;
                    Propagate $e_1 \ ^\star \ e_2$ to parent node;
                END
            UNLOCK the node
```

The evaluation template shows that a node is locked while being evaluated. This is necessary, because other evaluation threads may want to evaluate the node concurrently. If an event occurrence reaches the node and there is no suitable event occurrence available in the child lists, the new event occurrence is inserted into the corresponding child list. A child list is sorted with respect to the values of timestamps; the oldest event occurrence comes first and the newest comes last. In the event-operator-specific parts of the algorithms, a corresponding child list is searched for the first (that is, oldest) suitable event occurrence. This corresponds to the evaluation in the chronicle parameter context. Propagating an event occurrence to a parent node means accumulating the event parameters including the timestamp and activating the parent node's evaluation procedure.

The existence of a parameter restriction for a node implies the fulfillment of the *guard*-condition. For example, E_1 ; E_2 would be "and there is e_2 with $T(e_1) < T(e_2)$ and e_1.param = e_2.param" instead of "and there is e_2 with $T(e_1) < T(e_2)$".

8.1.2 Garbage Collection

Obsolete events can accumulate at specific nodes of a global event tree, namely in the right child list at a sequence node, in both child lists at a concurrency node, in the middle and right child lists at a negation node, and in the right child list at an iteration($^+$) node. Obsolete events should be garbage collected periodically, in order to avoid storage and performance overheads. Before garbage collecting any events, it must be guaranteed that the events are not needed. For example, in a sequence E_1 ; E_2 it must be guaranteed that there will be no $e_1 \in E_1$ with a smaller timestamp than a corresponding $e_2 \in E_2$. Depending on the structure of an overall global event tree this can be difficult, since e_1 may originate at different sites. Figure **??** illustrates this; e_1 may originate at all sites participating in the left subtree of (;). The garbage collection procedure respects the hierarchical structure of a global event tree. It starts at a root node and garbage collects the underlying tree recursively.

Algorithm 8.1.3 The following procedure *GarbageCollect_Tree* initiates the garbage collection process for a tree, starting with its *root* node. The procedure returns a timestamp, indicating until when the underlying tree has been garbage collected[1].

```
GarbageCollect_Tree(root): Timestamp
## is root currently being garbage collected ? ##
IF root.flag
    WAIT WHILE root.flag;        ## wait until garbage collection completes ##
```

[1]As in Modula-3, a RETURN or EXIT within a LOCK statement, LOCK mutex ... UNLOCK mutex, releases the mutex [Har92, page 242].

```
    RETURN root.last_gc;
ELSE
    LOCK root
        ## exceeds the garbage collection frequency the value of min_gc_frequency ? ##
        IF Timestamp.Now - root.last_gc < min_gc_frequency
            RETURN root.last_gc;
        ELSE
            root.flag := TRUE;        ## set flag for garbage collection ##
        END
    UNLOCK root
    ## fork threads for garbage collecting children nodes ##
    left := Fork a thread(GarbageCollect(root.leftchild));
    right := Fork a thread(GarbageCollect(root.rightchild));
    IF root.event_op = Negation
        Fork a thread(GarbageCollect(root.middlechild));
    END
    left_gc := WaitForThread(left);        ## join left and right threads ##
    right_gc := WaitForThread(right);
    gc_until := MIN{left_gc, right_gc};
    LOCK root        ## perform garbage collection until gc_until ##
        CASE root.event_op is
            Sequence =>
                Delete all $e$'s with $T(e)$ < gc_until from root.rightchild's list;
            Concurrency =>
                Delete all $e$'s with $T(e)$ < gc_until from root.leftchild's list;
                Delete all $e$'s with $T(e)$ < gc_until from root.rightchild's list;
            Negation =>
                Delete all $e$'s with $T(e)$ < gc_until from root.middlechild's list;
                Delete all $e$'s with $T(e)$ < gc_until from root.rightchild's list;
            Iteration($^+$) =>
                Delete all $e$'s with $T(e)$ < gc_until from root.rightchild's list;
        END
        root.last_gc := gc_until;
        root.flag := FALSE;        ## unset flag for garbage collection ##
    UNLOCK root
    RETURN root.last_gc;
END
```

Each *root* node has two variables, *flag* and *last_gc*; *flag* indicates whether the tree is currently being garbage collected and *last_gc* captures the timestamp of the last garbage collection. If the tree is currently being garbage collected or the last garbage collection was less than *min_gc_frequency* ago, the garbage collection does not proceed. Otherwise, it proceeds and *flag* is set. One garbage collection thread is then forked for each child node. The return values of the left and the right threads determine up to which timestamp the *root* node will be garbage collected.

Algorithm 8.1.4 The following procedure *GarbageCollect* initiates the garbage collection process for a subtree, starting with some *child* of a root node. The subtree may be degenerated to a single leaf. The procedure returns a timestamp, indicating until when the underlying subtree has been garbage collected.

GarbageCollect(child): Timestamp

```
IF child is leaf
    CASE leaf's event type originates at
        local site: RETURN (Timestamp.Now - $\Delta_{local}$);
        remote site: RETURN GarbageCollect_RemoteTree(child.event_type);
    END
ELSE ## child is node ##
    left := Fork a thread(GarbageCollect(child.leftchild));
    right := Fork a thread(GarbageCollect(child.rightchild));
    IF child.event_op = Negation
        Fork a thread(GarbageCollect(child.middlechild));
    END
    left_gc := WaitForThread(left);
    right_gc := WaitForThread(right);
    gc_until := MIN{left_gc, right_gc};
    LOCK child
        CASE child.event_op is
            Sequence =>
                Delete all $e$'s with $T(e)$ < gc_until from child.rightchild's list;
            Concurrency =>
                Delete all $e$'s with $T(e)$ < gc_until from child.leftchild's list;
                Delete all $e$'s with $T(e)$ < gc_until from child.rightchild's list;
            Negation =>
                Delete all $e$'s with $T(e)$ < gc_until from child.middlechild's list;
                Delete all $e$'s with $T(e)$ < gc_until from child.rightchild's list;
            Iteration($^+$) =>
                Delete all $e$'s with $T(e)$ < gc_until from child.rightchild's list;
        END
    UNLOCK child
    RETURN gc_until;
END
```

The procedure *GarbageCollect* is applicable to all children of a root node. If a child corresponds to a leaf, the procedure depends on whether the leaf relates to a local or a remote event. If it relates to a local event, the garbage collection stops and returns the current timestamp minus Δ_{local}, which corresponds to the *delay of local event detection* [**?**, page24] and identifies an upper bound for the time between the occurrence of a local event and its detection. Such a Δ can only be determined for local event detection, because there are no network delays and therefore there are no unknown primitive event occurrences up to the current point in time. If the leaf relates to a remote event, the request for garbage collection is forwarded to the remote site containing the event detector of the leaf's event type and the procedure *GarbageCollect_Tree* is applied to the root node of the corresponding event tree. If a child corresponds to a node, the garbage collection is done similarly to *GarbageCollect_Tree*; the procedure *GarbageCollect* is started for all children of the node and the node is garbage collected, depending on the return-values of the left and the right threads.

8.2 Synchronous Evaluation

Definition 8.2.1 (Synchronous evaluation) A global event tree 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 which have smaller timestamps have arrived.

Synchronous evaluation assumes the implementation of *FIFO network delivery*, that is, messages originating at any one site are delivered at any other site in the order they were generated [**?**]. FIFO network delivery can be achieved easily using TCP (Transmission Control Protocol) [Bac92] or any mechanism that numbers outgoing messages and delivers them accordingly.

Synchronous evaluation means that the evaluation of a node is delayed until all relevant events with smaller timestamps have arrived at the global event detector. Relevant events arrive from the sites relating to the siblings of the child node. There are no relevant events from the child node itself, because nodes are evaluated synchronously and because network delivery is FIFO; that is, all events from one node/leaf arrive in the order of their occurrence. In order to achieve this for leaves relating to remote global event trees, those global event trees have to be evaluated synchronously as well. The siblings of a child node may have other child nodes, which again have children, and so forth. Therefore, the existence of a relevant event depends recursively on numerous event reporting sites. Figure **??** illustrates this. The right child of (|) depends on all nodes in its subtree. In evaluating a composite event synchronously, every relevant primitive event that might participate in the evaluation of its parent node and other ancestors must be considered. Therefore, all event reporting sites in the subtree have to be checked for relevant events before an event occurrence can be detected at the parent node. In most cases, checking reveals that there are no relevant events. Occasionally there are relevant events which have been delayed because of failure.

8.2.1 Evaluation of Nodes

The evaluation algorithms for event operator nodes are presented next. Since events from one node/leaf arrive in the order of their occurrence, the garbage collection of obsolete events is incorporated into the evaluation algorithms.

The evaluation of conjunction(,)-, sequence(;)-, and concurrency(‖)-nodes is similar. The similarities are incorporated into an evaluation template. Operator-specific parts are stated thereafter. The evaluation of disjunction(|)-, negation(NOT)-, and iteration($^\star/^+$)-nodes differs considerably from asynchronous evaluation and involves the *SynchCheck* procedure for checking a subtree for relevant event occurrences up to a certain timestamp. The *SynchCheck* procedure is presented in Section 8.2.2.

Algorithm 8.2.2 The following procedures present the synchronous evaluation of conjunction(,)-, sequence(;)-, and concurrency(‖)-nodes:

```
Event_Op  LOCK the node
              IF $e_1$ is signalled
                  IF $E_2$'s list is not empty
                      ...BLOCK 2 ...
                  ELSE
                      Append $e_1$ to $E_1$'s list;
                  END
                  Set variable node.logical_sent;
              END
              IF $e_2$ is signalled
                  IF $E_1$'s list is not empty
                      ...BLOCK 2 ...
```

```
                ELSE
                    Append $e_2$ to $E_2$'s list;
                END
                Set variable node.logical_sent;
            END
        UNLOCK the node
```

```
$E_1$ , $E_2$     ...BLOCK 2 ...
                    $e_2$ = head of $E_2$'s list; delete head of $E_2$'s list;
                    Propagate $e_1$ , $e_2$ to parent node;
                  ...BLOCK 3 ...
                    ## in accordance ##
$E_1$ ; $E_2$     ...BLOCK 2 ...
                    ## GC: garbage collection of obsolete events ##
                    GC: Delete all $e_2$'s with $T(e_2) < T(e_1)$ from $E_2$'s list;
                    IF there is $e_2$ in $E_2$'s list with $T(e_1) < T(e_2)$
                        Delete $e_2$ from $E_2$'s list;
                        Propagate $e_1$ ; $e_2$ to parent node;
                    ELSE
                        Append $e_1$ to $E_1$'s list;
                    END
                  ...BLOCK 3 ...
                    ## in accordance, but no GC ##
$E_1 \| E_2$      ...BLOCK 2 ...
                    GC: Delete all $e_2$'s with $T(e_2) < T(e_1)$ from $E_2$'s list;
                    IF there is $e_2$ in $E_2$'s list with $T(e_1) \sim T(e_2)$
                        Delete $e_2$ from $E_2$'s list;
                        Propagate $e_1 \| e_2$ to parent node;
                    ELSE
                        Append $e_1$ to $E_1$'s list;
                    END
                  ...BLOCK 3 ...
                ## in accordance ##
```

Each node of a synchronously evaluated global event tree has a variable *logical_sent*, which captures the minimum value of the next event occurrence. This variable is set at the evaluation of an event operator node, either before the corresponding node is unlocked or before the evaluation procedure returns. The *SynchCheck* procedure uses *logical_sent* in order to determine whether the recursive scanning of a subtree can be stopped. The determination of the value for *logical_sent* is discussed later in this chapter.

The algorithms for conjunction(,)-, sequence(;)-, and concurrency(‖)-nodes are similar to asynchronous evaluation, except that synchronous evaluation includes the garbage collection of obsolete events. Moreover, since events from one node/leaf arrive in the order of their occurrence, events are being appended to child lists rather than being inserted, as in asynchronous evaluation.

Algorithm 8.2.3 The following procedure presents the synchronous evaluation of disjunction(|)-nodes:

```
$E_1 \mid E_2$    IF $e_1$ is signalled
                    LOCK the node
                      IF $E_2$'s list is not empty
                        Delete and propagate all $e_2$'s in $E_2$'s list
                          with $T(e_2) < T(e_1)$ to parent node;
◇                       IF there are $e_2$'s in $E_2$'s list with $T(e_2) \sim T(e_1)$
◇                         Delete and propagate all $e_2$'s in $E_2$'s list
◇                           with $T(e_2) \sim T(e_1)$ to parent node;
                          Set variable node.logical_sent;
                          Propagate $e_1$ to parent node; RETURN ;
                        ELSIF there are $e_2$'s in $E_2$'s list with $T(e_1) < T(e_2)$
                          Set variable node.logical_sent;
                          Propagate $e_1$ to parent node; RETURN;
                        ELSE
                          Append $e_1$ to $E_1$'s list;
                        END
                      ELSE
                        Append $e_1$ to $E_1$'s list;
                      END
                      Set variable node.logical_sent;
                    UNLOCK the node
                    *SynchCheck:* Check $E_2$'s subtree for event occurrences
                                  up to $e_1$'s timestamp;
                    LOCK the node
                      IF $e_1$ is still in $E_1$'s list
                        Set variable node.logical_sent;
                        Delete and propagate $e_1$ to parent node;
                      END
```

```
        UNLOCK the node
    END
    IF $e_2$ is signalled
        ## in accordance ##
    END
```

An event $e_1 \in E_1$ arriving at a disjunction(exclusive-or) node, $E_1 \mid E_2$, cannot be propagated immediately. First, all $e_2 \in E_2$'s with smaller timestamps than e_1 are propagated. After this, there may be e_2's left in E_2's list with concurrent or larger timestamps. Since concurrent events occur virtually "at the same time", there is no meaning to "the order of occurrence". Concurrent events are therefore propagated according to the "first come first served" principle. That means, all e_2's with concurrent timestamps are propagated before e_1. e_1 is also propagated, if there are e_2's with larger timestamps. In these cases, the evaluation of the node stops at this point. If there are no e_2's left in E_2's list with concurrent or larger timestamps, e_1 is appended to E_1's list, the node is unlocked, and the *SynchCheck* procedure is started in order to check whether there are e_2's which occurred earlier than e_1 but have not yet arrived at the node because of delay or failure. In the meanwhile, the disjunction node can be evaluated by other threads relating to newly arriving event occurrences. However, the *SynchCheck* procedure cannot be performed a second time before the first *SynchCheck* procedure returns. If this were possible, it could for example lead to the accumulation of a large number of *SynchCheck* procedure threads trying to check a remote site which has failed on a long-term basis. When the *SynchCheck* procedure returns, e_1 may have already been propagated due to arriving e_2's with larger timestamps than e_1. If this is not the case, that is, e_1 is still in E_1's list, the event is finally propagated.

The evaluation of a disjunction(inclusive-or) is similar to the evaluation of a disjunction (exclusive-or), except that the lines indicated by $\diamond$ are substituted with:

```
IF there is $e_2$ in $E_2$'s list with $T(e_2) \sim T(e_1)$
    Delete $e_2$ from $E_2$'s list;
    Propagate $e_1 \parallel e_2$ to parent node;
```

Algorithm 8.2.4 The following procedures present the synchronous evaluation of negation(NOT)- and iteration($\star$)-nodes:

E_1 ; NOT E_2 ; E_3

```
    IF $e_2$ is signalled
        LOCK the node
        Append $e_2$ to $E_2$'s list;
    UNLOCK the node
END
IF $e_1$ is signalled
    LOCK the node
        IF $E_3$'s list is not empty
            *GC:* Delete all $e_3$'s with $T(e_3) < T(e_1)$ from $E_3$'s list;
            IF there is $e_3$ with $T(e_1) < T(e_3)$
                Delete $e_3$ from $E_3$'s list;
                *GC:* Delete all $e_2$'s with $T(e_2) < T(e_1)$ from $E_2$'s list;
                IF there is $e_2$ in $E_2$'s list with $T(e_1) < T(e_2)$ and $T(e_2) < T(e_3)$
                    Set variable node.logical_sent;
                    RETURN;
                ELSIF there is $e_2$ in $E_2$'s list with $e_3 \; ; \; e_2$
                    Set variable node.logical_sent;
                    Propagate $e_1 \; ; \;$ NOT $e_2 \; ; \; e_3$ to parent node; RETURN;
                END
            ELSE
                Set variable node.logical_sent;
                Append $e_1$ to $E_1$'s list; RETURN;
            END
        ELSE
            Set variable node.logical_sent;
            Append $e_1$ to $E_1$'s list; RETURN;
        END
        Set variable node.logical_sent;
    UNLOCK the node
    *SynchCheck:* Check $E_2$'s subtree for event occurrences
                  up to $e_3$'s timestamp;
    LOCK the node
        IF there is no $e_2$ in $E_2$'s list with $T(e_1) < T(e_2)$ and $T(e_2) < T(e_3)$
            Set variable node.logical_sent;
            Propagate $e_1 \; ; \;$ NOT $e_2 \; ; \; e_3$ to parent node;
        END
    UNLOCK the node
END
IF $e_3$ is signalled
    ## in accordance, but no *GC* for $E_1$'s list ##
END
```

```
$E_1 \star E_2$    IF $e_1$ is signalled
                       LOCK the node
                           Append $e_1$ to $E_1$'s list;
                       UNLOCK the node
                   END
                   IF $e_2$ is signalled
                       SynchCheck: Check $E_1$'s subtree for event occurrences
                                   up to $e_2$'s timestamp;
                       LOCK the node
                           Set variable node.logical_sent;
                           Delete all $e_1$'s with $T(e_1) < T(e_2)$ from $E_1$'s list;
                           Propagate $e_1 \star e_2$ to parent node;
                       UNLOCK the node
                   END
```

A negation corresponds to a restricted sequence. After detecting the sequence $e_1 \ ; \ e_3$, it is checked whether there is any proof that the negation event cannot occur, that is, whether there is a corresponding e_2 in E_2's list which occurred in between e_1 and e_3. In this case, the negation is not detected. Further, it is checked whether there is an e_2 in E_2's list which occurred after e_3. In this case, the negation is detected. Otherwise, the node is unlocked and the subtree corresponding to the middle child E_2 is scanned for event occurrences up to e_3's timestamp. When the *SynchCheck* procedure returns, E_2's list is checked again for event occurrences lying in between e_1 and e_3. The negation is detected, if there are none. Note, that the detected sequence $e_1 \ ; \ e_3$ is not stored while the *SynchCheck* procedure runs. The reason is, that the negation could not be detected by another thread before the *SynchCheck* procedure returns; it could only be not-detected by another thread, that is, an e_2 arrives that occurred in between e_1 and e_3.

Before an iteration can be detected, it must be ensured that all $e_1 \in E_1$'s with smaller timestamps than an $e_2 \in E_2$ are available. Hence, the *SynchCheck* procedure is employed for scanning E_1's subtree. The evaluation of $E_1 \ ^+ \ E_2$ is a combination of $E_1 \ ; \ E_2$ and $E_1 \star E_2$ and is therefore not stated explicitly.

Each node has a variable *logical_sent*, which determines the minimum timestamp of the next event occurrence. The variable is set at the evaluation of an event operator node, either before the node is unlocked or before the evaluation terminates (that is, the evaluation procedure returns). The value of *logical_sent* depends on (i) the last event propagated to the parent node (ii) the events stored in the child lists and (iii) the event operator.

- Conjunction(,), Sequence(;), Concurrency(‖), Negation(NOT), Iteration($^+$): If both left and right child lists are empty, *logical_sent* corresponds to the timestamp of the last event propagated to the parent node. If there are events in one of the lists, *logical_sent* has the timestamp of the head of that list. Finally, if there are events in both lists, *logical_sent* corresponds to the latest timestamp of the heads of the lists.

- Disjunction(|), Iteration(*): *logical_sent* corresponds to the timestamp of the last event propagated to the parent node.

8.2.2 Synchronisation Procedure

There are several alternative methods for the synchronisation procedure *SynchCheck*. The following methods have been investigated:

Dummy Events Each local event participating in a global composite event is raised periodically as a dummy, if no "real" event has occurred for a certain length of time. Dummy events are propagated along the hierarchical structure of global event trees (evaluating, but not consuming any "real" events) and cause the occurrence of higher-level dummies, corresponding to global composite events. Dummies should be signalled periodically. If they are not, this indicates a delay or failure.

Token Passing Tokens are passed in a virtual ring of sites. When a token returns to its originator, this indicates that all sites were up and running at some specific point in time.

Request Messages When information on event occurrences is needed, a request is sent to the corresponding site and from there recursively to all sites which may participate in it. This procedure continues until it is confirmed that all relevant event occurrences have been signalled.

Although *dummy events* provide the information quickly, they cause an immense and unjustified overhead on event traffic and event processing. "Real" events will be detected later than necessary and a vast number of dummies will need to be garbage collected periodically. Hence, it does not seem to be sensible to base the synchronisation procedure on dummy events. On the other hand, *token passing* does not provide the information required by the synchronisation procedure. Local events may pass through numerous sites before they reach a certain node of a global event tree. Hence, although the site of event occurrence is up and running, the event may be stuck or delayed in a different site. Finally, *request messages* provide the information required by the synchronisation procedure when it is demanded, hence, not causing unnecessary event traffic and event processing overhead. Although the detection of global composite events may incorporate a slight delay, this delay can be limited in several ways. The synchronisation procedure developed in this section is based on the last method, *request messages*.

SynchCheck Algorithm

The synchronisation procedure and the garbage collection procedure (see Section 8.1.2) are similar. However, in contrast to garbage collection, a (sub)tree has to be evaluated only until it is guaranteed that no event occurrences up to a certain timestamp exist. This timestamp is determined by the initiator of the synchronisation procedure. Further, each node affected by a *SynchCheck* procedure must be locked for concurrent *SynchCheck* evaluations. The reason is that *SynchCheck* can originate at arbitrary nodes of a global event tree and not just at root nodes, as garbage collection.

Algorithm 8.2.5 The following procedure *SynchCheck* checks a (sub)tree starting with *child* for event occurrences up to the timestamp *checktime*. The subtree may degenerate to a single leaf.

SynchCheck(child, checktime)

```
IF child is leaf
   CASE leaf's event_type originates at
      local site: Wait until Timestamp.Now > checktime + $\Delta_{local}$
                  RETURN;
      remote site: SynchCheck_RemoteTree(child.event_type, checktime);
                   RETURN;
   END
ELSE ## child is node ##
   LOCK child
      WAIT WHILE child.flag; ## wait until other thread completes SynchCheck ##
      child.flag := TRUE;       ## set flag for SynchCheck ##
   UNLOCK child
   ## is the min timestamp of the next event occurrence larger than checktime ? ##
```

```
  IF child.logical_sent $\geq$ checktime
    child.flag := FALSE;      ## unset flag for SynchCheck ##
    RETURN;
  ELSE      ## perform SynchCheck ##
    CASE child.event_op is
      Conjunction, Sequence, Concurrency, Negation =>
        left := Fork a thread(SynchCheck(child.leftchild, checktime));
        right := Fork a thread(SynchCheck(child.rightchild, checktime));
        IF child.event_op = Negation
          middle := Fork a thread(SynchCheck(child.middlechild, checktime));
        END
        Wait for left or right thread to return;
        IF there are no event occurrences in the corresponding child list
          with timestamps smaller than checktime THEN
          Alert other threads;
          Synch_Completion();
        ELSE
          Wait for other thread(s) to return;
          Synch_Completion();
        END
      Disjunction =>
        left := Fork a thread(SynchCheck(child.leftchild, checktime));
        right:= Fork a thread(SynchCheck(child.rightchild, checktime));
        Wait for left and right threads to return;
        Synch_Completion();
      Iteration($\star$) =>
        left := Fork a thread(SynchCheck(child.leftchild, checktime));
        right:= Fork a thread(SynchCheck(child.rightchild, checktime));
        Wait for right thread to return;
        IF there are no event occurrences in the corresponding child list
          with timestamps smaller than checktime THEN
          Alert other thread;
          Synch_Completion();
        ELSE
          Wait for left thread to return;
          Synch_Completion();
        END
    END
  END
END
```

Synch_Completion()

```
LOCK child
    child.logical_sent := MAX{child.logical_sent, checktime};
    child.flag := FALSE;
UNLOCK child
RETURN;
```

The procedure *SynchCheck* is applied to one or more children of a node. The procedure depends on whether a child corresponds to a leaf or a node. If it corresponds to a leaf, the procedure depends on whether the leaf relates to a local or a remote event. In the first case, the procedure returns as soon as the current timestamp exceeds the value of *checktime* plus Δ_{local} (the delay of local event detection). This guarantees that there are no local event occurrences up to *checktime*. Since *checktime* is an earlier timestamp and Δ_{local} is small, there is usually no waiting involved. If the leaf relates to a remote event, the request for synchronous checking is forwarded to the remote site containing the event detector of the leaf's event type and the procedure *SynchCheck* is applied to the root node of the corresponding event tree. If a child corresponds to a node, the procedure can only proceed if the node is not currently being *SynchCheck*ed; *flag* is set as soon as the subtree becomes available for synchronous checking. Maybe at that point, event occurrences with larger timestamps than *checktime* have already been propagated to the parent node. This is revealed by the variable *logical_sent*. In this case, it is not necessary to proceed scanning the subtree for smaller event occurrences. There are none, since events from one node/leaf are signalled in the order of their occurrence. If *logical_sent* is smaller than *checktime*, there may be event occurrences with smaller timestamps in the corresponding subtree. In this case, one thread is forked for each child of the node, performing *SynchCheck* in the corresponding subtrees concurrently. The further evaluation depends on the event operator. For a conjunction, sequence, concurrency, and negation the return of the left or right thread is awaited. If no suitable event occurrence has arrived at the node in the meanwhile, the procedure stops; it is impossible to detect one of these events without a suitable event occurrence from the left or the right child. For a disjunction, the return of both *SynchCheck* threads is awaited, because only one event occurrence is necessary in order to detect a disjunction. Finally, for an iteration($\star$), if the right thread reveals that there are no suitable event occurrences, the procedure stops. Otherwise, the return of the left thread is awaited.

8.3 Summary

Two different evaluation policies were introduced in this chapter; asynchronous and synchronous evaluation represent extreme possibilities for dealing with delayed events at the evaluation of event operator nodes.

Evaluating a global event tree asynchronously means evaluating nodes instantly on the arrival of suitable event occurrences, whereas delayed events are not considered. Hence, events are not evaluated in the order of their occurrence, that is, do not respect $2g_g$-restricted temporal order (see Definition 6.2.2). On the other hand, event detection is not blocked by delayed events and is therefore faster. For most event operators asynchronous evaluation results in inconsistency with respect to the parameter context or incompleteness of parameter computations. However, for a negation(NOT) it can lead to "wrong" event occurrences, because in E_1 ; NOT E_2 ; E_3 a relevant event $e_2 \in E_2$ may be delayed. If this is not acceptable, a negation must be evaluated synchronously.

Asynchronous evaluation is suitable for real-time applications [**?**, Chapter 16] or any applications that require fast response times; it is crucial that an event is detected as quickly as possible, but it is not crucial that the event parameters reflect the system-wide chronological order of event occurrences.

In synchronously evaluated global event trees, each node is evaluated regarding the $2g_g$-restricted temporal order of all events which may participate in an occurrence. The evaluation of a node blocks until all corresponding sites have been checked for relevant event occurrences. This is done using the synchronisation procedure *SynchCheck*. Although synchronous evaluation guarantees that events relating to specific nodes are evaluated in the order of their occurrence, the evaluation can block long-term, if there are site failures, network partitioning, or network congestion.

Synchronous evaluation is suitable for applications requiring a high degree of consistency and reliability, such as the traditional database applications of banking and warehousing.

As alternatives to asynchronous and synchronous evaluation, a whole range of evaluation policies is possible which lie in between the two; a global event tree could be evaluated synchronously, specifying an upper time-limit for events to be considered. For example, *synchronous within two minutes* specifies that a global event tree is evaluated according to synchronous semantics, but that the evaluation of a node continues after two minutes blocking.

Chapter 9

Prototype Implementation and Evaluation

The goal of the prototype implementation is to realise the detection of global composite events in distributed systems. For this purpose, it is not necessary to support the full extend of primitive event specification and detection; the essential behaviour is the signalling of primitive events and specifically, of their event type and timestamp. This information will be simulated by *event simulators*.

The prototype implementation is concerned with

- the structure and handling of timestamps according to the simplified semantics, as discussed in Sections 6.4 and 6.5.
- the asynchronous and synchronous detection of global composite events composed of simulated primitive and/or detected global composite events and event operators conjunction(,), disjunction(|), sequence(;), concurrency(‖), negation(NOT), and iteration($\star$), as discussed in Chapters 7 and 8.

The prototype implementation is not concerned with

- the realisation of a distributed database system for the specification and detection of data manipulation events and transaction events.
- the employment of a programmable timer interface for the specification and detection of time events.
- the handling of complex event parameters. The only parameter used for event detection is the timestamp of an event occurrence. Other parameters have no

influence on the detection process.

9.1 Programming Environment

The mechanisms for detecting global composite events in distributed systems have been implemented using *Modula-3 for Network Objects* [BNOW94]. Modula-3 for Network Objects is a distributed programming system extending *Modula-3* [Har92], where communication over a network is done using network objects. A network object is an object whose methods can be invoked over a network. The program containing a network object is called the *owner* of the network object and the program using it is called the *client*. An important feature of Modula-3 for Network Objects is that a *network object pointer* can be passed as an argument or result between sites. This provides a more powerful mechanism than ordinary RPC. Moreover, Modula-3 for Network Objects provides *distributed garbage collection*.

The prototype has been implemented and tested on a number of Sun SPARC workstations at the University of Cambridge Computer Laboratory, connected by an Ethernet network. FIFO network delivery can be assumed in this environment, because communication is based on TCP (Transmission Control Protocol) [Bac92].

9.2 Architecture

The prototype includes two processes running independently of each other on each site in the distributed test environment: an *event simulator process* and an *event detector process*. In a test environment of n sites, there are $2 \times n$ processes, n event simulators and n event detectors.

An *event simulator* simulates primitive event occurrences. There are two important points to consider: *which* primitive events to simulate and *when*. The essential information captured in a primitive event occurrence is the event type and the timestamp. Since primitive event types are site-related, a primitive event type is modelled as a tuple consisting of the name of a corresponding event simulator site and a number distinguishing it from the other types at that site. The timestamp denotes the time of an event occurrence and in this case the time of the event simulation. The timestamp is allocated after the simulation of a primitive event occurrence, before it is signalled to the event detectors. Between simulating and signalling two primitive event occurrences, the event simulator pauses for a certain time period. This time period can be determined by the user.

An *event detector* receives event occurrences from multiple sites via different input ports. In the current implementation, there is one input port for each event reporting site[1]. Since network delivery is FIFO point-to-point, the event occurrences arrive at a single port in the order in which they were sent. However, event occurrences arrive concurrently at different ports. For each signalled event occurrence, a thread is forked to evaluate the global event trees at the local site. The threads synchronise at the nodes of the global event trees. This means, if two event occurrences arrive at a node simultaneously, one thread is evaluated while the other one is waiting. Different nodes can, however, be evaluated concurrently. Detected global composite events are displayed to the user and/or are signalled to registered event detector sites for further evaluation.

Figure **??** illustrates the system architecture with four sites. Event simulators signal primitive event occurrences to sites via the corresponding input ports. Each input port receives the simulated primitive and detected global composite event occurrences from a single generating site. The event detectors evaluate incoming event occurrences concurrently. Detected global composite events can be resignalled to any event detector site.

9.3 Data Structures

There is a distinction between `event type`s and `event occurrence`s. Event occurrences are characterised by their `timestamp` and other event parameters. The event occurrences flowing around in an event graph are stored in `lists of` event `occurrences`. An event graph consists of `leaf`s, `node`s, and `root`s. In addition to an event graph, each site has two hash tables: an `event table` and a `site port table`. Figure **??** shows the dynamic data structures at an event detector site. Each event simulator site has one hash table: an `observer_register`.

9.3.1 Event Type

An `event_type` is defined as an object type with two fields: `site` and `event_nr`. `site` is a string denoting the site from which the event type originates and `event_nr` is a cardinal identifying the event type in relation to `site`. Since primitive events are simulated, no further information on event types is necessary. For example,

[1]In large-scale distributed systems, there will be one input port for multiple event reporting sites, because of scalability.

`pelican.cl.cam.ac.uk.3` determines event type number `3` originating at site `pelican.cl.cam.ac.uk`.

9.3.2 Global Time

A `global time` is defined as an object type with the fields `year`, `month`, `day`, and `ticks`, where `year` and `ticks` are cardinal numbers and `month` and `day` are subranges of cardinal numbers. `ticks` denotes the global clock ticks (ticks of granularity g_g) since midnight on the same day. For testing purposes, additional fields `hour`, `minute`, `second`, and `fraction` are kept.

9.3.3 Timestamp

A `timestamp` is defined as an object type with three fields: `base`, `interval`, and `offset`. This representation of a timestamp corresponds to the simplified semantic model, assuming no two primitive event occurrences at the same site within one global clock tick (see Section 6.4). `base` depicts a global time read at an arbitrary site in the distributed system. `interval` is a boolean and states whether the distance between the base and the limit of the timestamp is 0 (`interval` = FALSE) or 1 (`interval` = TRUE) global clock tick. Finally, `offset` denotes a list of tuples `(host,bound)` indicating which sites participate in the timestamp and to which global time they relate (to the base or to the limit of the timestamp).

9.3.4 Event Occurrence

An `event_occurrence` is defined as an object type with three fields: `time_occur`, `time_detect`, and `parameters`. `time_occur` and `time_detect` are both timestamps denoting the time of event occurrence and the time of event detection. Other parameters are captured in the field `parameters` which is simply a list of pointers. Each `event_occurrence` relates to a specific event type. However, the event type is not stored explicitly, but is implicit depending on the position of an event occurrence in an event graph.

9.3.5 List of Event Occurrences

At runtime, event occurrences accumulate at the nodes of event trees and are stored in `lists of occurrences`. A `list of occurrences` is a list of `event_occurrence` objects and relates to a specific event type.

9.3.6 Leaf

A `leaf` of an event graph is defined as an object type with the following attributes: `event_type` indicates which event occurrences are to be inserted into the leaf, `parent` is a pointer to the parent node in the event graph, and `position` determines whether the leaf corresponds to a left, a middle, or a right child.

9.3.7 Node

A `node` of an event graph is defined as an object type with the following attributes: `parent` is a pointer to the parent node in the event graph, `operator` states the event operator, and `position` determines whether the node is a left, a middle, a right child, or a root. Further each node contains information corresponding to the `left` and the `right` child. Negation nodes have a third `middle` child. This information consists of a `list of occurrences` and a pointer to the child node.

Synchronous Evaluation

Nodes which are synchronously evaluated have additional attributes `logical_sent` and `flag`, denoting the minimum timestamp of the next event occurrence and the flag employed in *SynchCheck* respectively.

9.3.8 Root

A `root` is a subtype of `node` with two additional attributes `event_type` and `registered_sites`. `event_type` defines the event type of detected occurrences at this node and `registered_sites` indicates which sites have registered their interest in the event type.

Asynchronous Evaluation

Roots which are asynchronously evaluated have additional attributes `last_gc` and `flag`, capturing the time of the last garbage collection and the flag used in *Garbage-Collect_Tree* respectively.

9.3.9 Event Table

An `event_table` is a hash table mapping event types onto lists of leaves and/or root nodes. All leaves and roots relate to the local site. There is one event table at each site, serving as the interface for accessing the local event detection mechanisms.

9.3.10 Port

A `port` is a network object with three methods: `signal_event`, `request_gc`, and `request_sy`. Primitive and composite events are signalled at a site with the help of the `signal_event` method, whereas the other two methods are used to forward requests for garbage collection and synchronous checking respectively to the site.

9.3.11 Site Port Table

A `site_port_table` is a hash table mapping sites to imported and exported port network objects. One network object is exported for each site which may signal event occurrences and one network object is imported for each site which has registered an interest into detected event occurrences. There is one site port table at each site.

9.3.12 Observer Register

An `observer_register` is a hash table employed at an event simulator site. It maps event types onto lists of imported port network objects.

9.4 System Setup and Initialisation

There is a single *coordinator process* which coordinates the system setup and initialisation. In order to perform system setup and initialisation, each event detector process exports a network object `InitCEDetect` having two methods `setup` and `connect` and each event simulator process exports a network object `InitPESim` having two methods `setup` and `start_simulate`. The coordinator process imports these network objects and calls the corresponding methods in turn.

9.4.1 First Step: `InitCEDetect.setup`

`InitCEDetect.setup` has one parameter, a string denoting a file accessible by the corresponding detector process. This file contains the setup information for the event detector, namely

(1) the evaluation policy *asynchronous* or *synchronous*

(2) the global event trees to be evaluated, and

(3) the event detectors from which simulated primitive and detected composite event occurrences are signalled.

(1) determines the evaluation policy applied to the event graph at an event detector process. If both evaluation policies are to be implemented at the same site, two event detector processes are utilised. (2) relates to setting up the dynamic data structures shown in Figure **??** and (3) is concerned with exporting port network objects and storing them in the `site_port_table`.

9.4.2 Second Step: `InitPESim.setup`

`InitPESim.setup` has one parameter, a string denoting a file accessible by the corresponding simulator process. This file contains

(1) the simulated event types and

(2) the event detectors to which the simulated event occurrences are signalled.

For each simulated event type mentioned in (1), there is an entry in the `observer_register`. A corresponding value consists of a list of imported port network objects, relating to the event detectors to which simulated event occurrences of that type are signalled.

9.4.3 Third Step: `InitCEDetect.connect`

The `InitCEDetect.connect` method completes the setup process at an event detector. In `InitCEDetect.setup` one port network object was exported for each site signalling event occurrences. However, detected global composite events are resignalled to registered event detector sites. Hence, corresponding port network objects are imported and stored in the `site_port_table`. Note, that `InitCEDetect.connect` cannot be merged with `InitCEDetect.setup`, because importing port network objects implies that the corresponding event detectors are already installed.

9.4.4 Fourth Step: `InitPESim.start_simulate`

At this point, all event simulators and event detectors are set up and initialised. Hence, the event simulation and event detection can proceed. The `InitPESim.start_simulate` method of each event simulator is called in turn, passing a string parameter denoting a file accessible by the corresponding simulator process. This file contains information on the simulation process as such.

9.5 Runtime Behaviour of the Event Simulator

The call of the method `InitPESim.start_simulate(name:TEXT)` triggers the simulation of primitive event occurrences at the corresponding event simulator site. The file denoted by `name` contains the information on the event simulation process and has the following form:

$$event_1 \quad pause_1 \quad event_2 \quad pause_2 \quad event_3 \quad \ldots$$

$event_1$ is a cardinal number relating to the event type at the local site. For example, if the event simulator site is `pelican.cl.cam.ac.uk` and $event_1$ is `3`, an occurrence of primitive event type `pelican.cl.cam.ac.uk.3` is simulated. Simulating an event occurrence means determining the event type and the timestamp relating to the current point in time and signalling the corresponding values to all event detector sites which are registered under event type in the `observer_register`. $pause_1$ is a real number denoting a time period (in seconds). The event simulator pauses for $pause_1$ seconds after simulating the event occurrence relating to $event_1$ and before simulating the event occurrence relating to $event_2$. This procedure is repeated for all $event_n$ and $pause_n$ until the end of file is reached.

9.6 Runtime Behaviour of the Event Detector

9.6.1 Signalling Event Occurrences

Simulated primitive and detected global composite event occurrences are signalled at an event detector site with the help of the `signal_event` method of a corresponding `port` network object. On the arrival of an event occurrence, the `event_table` at the local site is consulted in order to find an entry for the corresponding event type. If there is no entry or the list of leaves is empty, the event type does not participate in a global composite event and the event occurrence can therefore not be used. Otherwise, the event occurrence is injected into all leaves in the list of leaves. The ordering of a list of leaves reflects rule priorities; the first leaf corresponds to a rule with a higher priority than the second leaf and so on (see Section 2.1). An injected event occurrence does not reside in a leaf, but is directly propagated to the parent node.

9.6.2 Evaluating Nodes

An event occurrence arriving at a node is evaluated according to the evaluation procedures presented in Chapter 8.

Asynchronous evaluation requires nodes to be evaluated instantly, without considering delayed events. *Garbage collection* of obsolete events accumulating at sequence-, concurrency-, negation-, and iteration($^+$)-nodes, is performed separately. Garbage collection starts with the root of a global event tree, when the corresponding global composite event is detected and the last garbage collection was more than one minute ago[2]. If a garbage collected leaf relates to a remote event, the procedure *GarbageCollect_RemoteTree* in Algorithm 8.1.4 involves calling the `request_gc` method of a corresponding `port` network object. This method has one attribute, the `event_nr` of the leaf's `event type`, which determines the global event tree to be garbage collected next. Asynchronous evaluation of nodes has been implemented as shown in Section 8.1.

Synchronous evaluation requires that all relevant event occurrences are considered in the order of their occurrence, when a node is evaluated. Depending on the event operator and the available event occurrences (the newly arriving event occurrence and the event occurrences stored in the node's child lists), a node is evaluated instantly or delayed. Disjunction-, negation-, and iteration-nodes are evaluated delayed, if the *SynchCheck* procedure is involved. Otherwise, they are evaluated instantly. Conjunction-, sequence-, and concurrency-nodes are always evaluated instantly. If a *SynchCheck*ed leaf relates to a remote event, the procedure *SynchCheck_RemoteTree* in Algorithm 8.2.5 involves calling the `request_sy` method of a corresponding `port` network object. The implementation of `request_sy` corresponds to the implementation of `request_gc`. Synchronous evaluation of nodes has been implemented as shown in Section 8.2.

Both *asynchronous* and *synchronous evaluation* involve procedures regarding the handling of timestamps, as discussed in Sections 6.4 and 6.5. The procedure `Timestamp.Get` allocates a new timestamp relating to the current point in time, `Timestamp.Compare` compares two timestamps and indicates their temporal relationship (see Definition 6.4.5), and `Timestamp.Join` joins two concurrent or unrelated timestamps (see Definition 6.4.7).

[2]This value relates to the variable *min_gc_frequency* in Algorithm 8.1.3 which can be set to any value.

9.6.3 Deriving Event Parameters

When an event occurrence is detected at any node of a global event tree, it is propagated to the parent node or displayed to the user (in the case of a root node). "It" means its event parameters: `time_occur`, `time_detect`, and `parameters` (see Section 9.3.4). `time_occur`, the time of event occurrence, represents the main timestamp of an event occurrence, which is essential in the whole event detection process. It is derived according to the semantics of event operators (see Definition 6.5). `time_detect` denotes the time of event detection and is allocated just before an event occurrence is propagated to the parent node or displayed to the user. `time_detect` is an auxiliary timestamp, used to determine the *delay of global event detection*, that is, the time between the occurrence of an event and its detection. It is a measure of the system's performance. `parameters` is a list of pointers to the operand event occurrences of a detected event occurrence. There are two pointers for a conjunction, sequence, concurrency, and negation, one pointer for a disjunction(exclusive-or), one or two pointers for a disjunction(inclusive-or) and one or more pointers for an iteration (depending on the number of left child events). Since the operand event occurrences may be composite themselves, the parameters form a tree.

9.6.4 Detecting Event Occurrences

When an event occurrence is detected at the root node of a global event tree, a corresponding global composite event is raised. The `event_type` attribute of the root node determines the event type of the raised global composite event and the `registered_sites` attribute indicates which sites have registered their interest in this event type. The `site_port_table` at the local site is then consulted in turn, in order to find a reference to the `port` network object at the corresponding remote site. The event occurrence is signalled to that site by calling the `signal_event` method of the `port` network object.

Detected event occurrences have a `parameters` attribute containing a hierarchical structure of pointers to constituent event occurrences. Since Modula-3 for Network Objects allows the use of network object pointers to be passed as arguments or results between sites (see Section 9.1), the `parameters` attribute does not have to be marshalled, but can be transmitted as such. Figure **??** shows an instance of a global event tree at event detector site 1 with network object pointers to sites 2 and 3. The use of network object pointers has numerous advantages:

- It minimises the copying of data. Event occurrences often have extensive parameters. Also, event occurrences of a certain event type are often used in more than one event detector. Transmitting the reference to an event occurrence saves marshalling and copying the event occurrence with all its parameters.
- Abstraction: it allows different local representations of objects and, in particular, of event occurrences at different sites, and can therefore be applied in heterogeneous systems.
- Audit logging: each site can keep an audit trail of events structured for local purposes. The global event trail will be a structured path through these local logs.
- Traceability: the full structure can be traversed by any observer. This gives the possibility of global logging and of alternative semantics.

9.6.5 Concurrency

The dynamic data structures at an event detector site are evaluated concurrently by different threads:

- one thread for all event occurrences signalled at one of n `port` network objects
- one thread for each event occurrence injected into a leaf
- one thread for each event occurrence propagated to a parent node
- one thread for each event occurrence signalled to a remote event detector site
- *asynchronous evaluation:* one thread for each branch in a garbage collected (sub)tree
- *synchronous evaluation:* one thread for each branch in a *SynchCheck*ed (sub)-tree

The threads evaluating global event trees synchronise at nodes, that is, the evaluation of a node is a critical region and the node is locked while being evaluated. Another measure for synchronisation is the employment of flags for garbage collection (asynchronous evaluation) and *SynchCheck*ing (synchronous evaluation). Only one such thread can access a corresponding (sub)tree at any one time.

The extensive use of threads allows the concurrent evaluation of the global event trees at an event detector site. This has the following advantages:

- It allows different nodes to be evaluated concurrently. This is especially useful in synchronous evaluation, because the evaluation of a node can block for a long time if an event reporting site has failed. Other nodes, which are not influenced by the failed site, can be evaluated in the meanwhile.
- It allows garbage collection and *SynchCheck*ing to run in the background; hence, not influencing normal evaluation activity.
- It reflects the concurrency inherent in distributed systems.

9.7 Evaluation

The goal of the prototype evaluation is to examine the differences between asynchronous and synchronous evaluation. There are two aspects:

- *What* event occurrence is detected?
- *When* is an event occurrence detected?

The first aspect relates to a detected event occurrence as such and to its structure as a composition of constituent event occurrences. The second aspect denotes the delay of global event detection and is a measure of the system's performance. The differences between asynchronous and synchronous evaluation become clear at the evaluation of disjunction(|)-, negation(NOT)-, and iteration($^\star/^+$)- nodes. The other event operators are handled similarly in the two evaluation modes and will therefore behave in the same way in each.

In the following, three tests are described relating to global composite events with event operators disjunction, negation, and iteration. Larger network delays were enforced for parts of the second and third test, in order to present the behaviour of asynchronous and synchronous evaluation in critical situations (i.e. site failures and network congestion). Therefore, the test results are not a representative cross-section of normal runtime behaviour. The global clock granularity corresponds to $g_g = 1/10sec$. The event traces for parts of the second and third test are given in Appendix A.

9.7.1 Test One

Figure **??** shows four global event trees evaluated at three sites: pelican, kookaburra, and osprey. There are two global event trees at pelican (relating to events `pelican.10` and `pelican.20`), one at kookaburra (relating to event `kookaburra.10`), and one at osprey (relating to event `osprey.10`). When an event `pelican.20` is detected, it is signalled to osprey, where it causes the detection of `osprey.10`, which causes the detection of `kookaburra.10`, which finally causes the detection of `pelican.10`. Three different tests were performed regarding simulated primitive events `pelican.1`, `kookaburra.1`, and `osprey.1` respectively. The test results, representing the differences between time of occurrence and time of detection (in global clock ticks), are illustrated in Table 9.1[3]. The first and third column relate to measurements taken during the day and the second and fourth column relate to measurements taken at night.

	Asynchronous		Synchronous	
pelican 1	2	2	6	6
	1	1	4	4
	2	2	4	4
	1	1	3	4
	1	1	3	4
kookaburra 1	2–4	3–5	4–6	3–5
	2–4	3–5	3–6	3–6
	2–4	2–4	4–6	4–6
	1–4	3–8	3–5	7–10
	2–5	2–7	7–9	6–8
osprey 1	1–3–5	1–2–4	1–4–6	1–3–9
	1–2–4	0–2–4	1–3–5	2–3–5
	4–8–10	1–3–5	4–5–7	3–4–6
	1–3–5	1–3–6	3–4–7	2–3–5
	1–3–5	0–2–7	2–3–8	2–3–6

Table 9.1: Delay of Global Event Detection

In the first test, the simulation of `pelican.1` causes the detection of `pelican.10`. The results confirm that asynchronous evaluation is considerably quicker than syn-

[3]Since the time of occurrence and the time of detection of an event may originate at different sites, the synchronisation of local clocks has to be taken into account. In the test environment, the precision Π of the synchronised local clocks is in the order of $\Pi = 1/100sec$ and therefore small in comparison with the global clock granularity $g_g = 1/10sec$.

chronous evaluation. In synchronous evaluation, the whole subtree relating to `kookaburra.10` is *SynchCheck*ed before detecting `pelican.10`, whereas `pelican.10` is detected instantly in asynchronous evaluation.

In the second test, the simulation of `kookaburra.1` causes the detection of `kookaburra.10` (first value in m–n) and `pelican.10` (second value in m–n). The results show that the differences between asynchronous and synchronous detection of `kookaburra.10` are less significant. The reason is that the *SynchCheck*ed subtree is flatter than in the previous test. The delay in the detection of `pelican.10` incorporates the delay in the detection of `kookaburra.10` and the transmission of `kookaburra.10` to pelican. The delay due to *SynchCheck* at `pelican.10` is negligible, because the subtree consists of a single leaf at the local site.

In the third test, the simulation of `osprey.1` causes the detection of `osprey.10` (first value in m–n–o), `kookaburra.10` (second value in m–n–o), and `pelican.10` (third value in m–n–o). The differences in asynchronous and synchronous detection of `osprey.10` are again less significant, because the *SynchCheck*ed subtree is flatter.

Note, that this test considers a worst case scenario, because the global event trees consist of disjunction nodes only. For disjunction nodes, the *SynchCheck* procedure has to wait for both forked *SynchCheck* threads to return (see Algorithm 8.2.5). This is not the case for other event operators. For example, if `kookaburra.10` is a sequence node, *SynchCheck* returns when the *SynchCheck* thread relating to the left child `kookaburra.1` returns and there are no relevant event occurrences in the left child list. In general, *SynchCheck* stops as soon as it finds evidence that there are no relevant event occurrences.

9.7.2 Test Two

Test two investigates the problems regarding the detection of negations. Figure **??** shows that the middle child of `pelican.10` relates to the remote subtree of `kookaburra.10`, which relates to `osprey.10`. On the detection of a sequence between `pelican.1` and `pelican.2`, the asynchronous evaluation checks the available middle child events and signals `pelican.10`, if there is no `kookaburra.10` in between. On the other hand, the synchronous evaluation *SynchCheck*s the subtree relating to `kookaburra.10`, to determine whether there are relevant events.

Figures **??** and **??** present the test results for the asynchronous and synchronous evaluation respectively[4]. In the first two cases of asynchronous evaluation and in

[4]Vertical lines indicate the occurrence of events and bullets indicate the detection of events.

synchronous evaluation, larger network delays were enforced by manually increasing the time period between the detection and the signalling of an event.

`osprey.10` is detected shortly after the occurrence of `osprey.2` and transmitted to kookaburra, where it causes the detection of `kookaburra.10`. `kookaburra.10` is then transmitted to pelican. In the first two cases of asynchronous evaluation, `kookaburra.10` arrives after the occurrence of `pelican.2` and the detection of `pelican.10`. Hence, the negation is detected, although there is a `kookaburra.10` which occurred in between `pelican.1` and `pelican.2` (the time of occurrence is that of `osprey.2`). In synchronous evaluation, no negation is detected, because the evaluation blocks until `kookaburra.10` has arrived. In the third case of asynchronous evaluation, the delays are less significant and `kookaburra.10` arrives at pelican before the occurrence of `pelican.2`. Hence, the negation is not detected.

The results confirm that different event occurrences may be detected for a negation in asynchronous and synchronous evaluation; in one case an event occurrence is detected, in the other case an event occurrence is not detected. In asynchronous evaluation, negation is detected with respect to the relevant events arriving at the site "in time", whereas in synchronous evaluation, negation is detected with respect to all relevant events in the system.

9.7.3 Test Three

Test three investigates the problems regarding the detection of iterations. In Figure **??**, the left child of `pelican.10` relates to the remote event `kookaburra.10`. On the detection of `pelican.1`, asynchronous evaluation signals the occurrence of `pelican.10` with all available `kookaburra.10`s. Synchronous evaluation employs the *SynchCheck* procedure beforehand in order to ensure that all relevant `kookaburra.10`s have arrived.

Figures **??** and **??** present the test results for asynchronous and synchronous evaluation respectively. In the first two cases of asynchronous and synchronous evaluation, larger network delays were enforced by manually increasing the time period between the detection and the signalling of an event.

At the first occurrence of `pelican.1`, there are no `kookaburra.10`s and thus `pelican.10` is detected with no left child events. However, at the second occurrence of `pelican.1`, there are three `kookaburra.10`s. In the first case of asynchronous evaluation, the third `kookaburra.10` arrives at pelican after the occurrence of `pelican.1`; hence, `pelican.10` is detected with only two out of three `kookaburra.10`s. In the second case, the second `kookaburra.10` is delayed as well and there is only one left child event at the detection of `pelican.10`. Finally in the third case, the delays are less significant and all three `kookaburra.10`s arrive in time. In synchronous evaluation, the second `pelican.10` is not detected until all three `kookaburra.10`s have arrived. The detection of the first `pelican.10` takes a little longer, because the subtree relating to `kookaburra.10` is *SynchCheck*ed beforehand.

The results confirm that different event occurrences may be detected for an iteration in asynchronous and synchronous evaluation; the number of left child events may differ. In asynchronous evaluation, a relevant left child event may arrive too late to contribute to a composite event occurrence.

9.8 Summary

This chapter described the prototype implementation realising the detection of global composite events in distributed systems. Primitive event occurrences are simulated by event simulators placed at all sites of a distributed test environment, and signalled to all registered event detectors. Event detectors evaluate global event trees either asynchronously or synchronously. Detected occurrences of global composite events are displayed to the user and/or are signalled to registered event detectors for further evaluation. In order to evaluate the prototype, several tests were performed illustrating the different runtime behaviour of asynchronous and synchronous evaluation.

Chapter 10

Conclusions

This dissertation presented an approach to monitoring the behaviour of distributed systems in terms of events. A summary of the main contributions and an outlook on further work is given in this chapter.

10.1 Summary

An approach to event-driven monitoring of distributed systems must provide the following functionality:

- It must support the specification of primitive and composite events. In order to monitor the external and internal behaviour of distributed systems, primitive events must relate to physical time and sensor readings as well as to occurrences within the system (e.g. within a database or an application program). Further, the event operators used to construct composite events must be applicable to events at local and at remote sites.

- It must be semantically sound. A primitive or composite event expression describes a specific system behaviour. The semantics must be well-defined, that is, must determine clearly what this behaviour is.

- It must provide algorithms for the detection of primitive and composite events at system runtime. The detection of primitive events is straightforward. The detection of composite events is an incremental process, based on some underlying computational model. The algorithms for the detection of composite events must take account of the special characteristics of distributed systems.

The research in this dissertation was motivated by the observation that current approaches to event-driven monitoring of distributed systems do not provide this functionality. The commercial standards COM (Component Object Model) [?] and CORBA (Common Object Request Broker Architecture) [?] introduce basic event services. However, they do not support general composite events. Work in active database systems demonstrates sophisticated methods for specifying and detecting primitive and composite events in centralised systems. But, distributed systems are not considered. Distributed debugging systems employ primitive and composite events for monitoring distributed computations; the causal relationship of primitive events is monitored in order to locate the cause of errors. Hence, distributed debugging systems monitor the internal system behaviour only.

This dissertation has proposed an approach to event-driven monitoring of distributed systems which provides the full functionality of event specification, event semantics, and event detection. Before these issues could be addressed, it was necessary to review the research areas of active database systems and distributed debugging systems, and to assess their applicability for monitoring the internal and external behaviour of distributed systems.

Event specification was the subject of Chapter 5. Different kinds of primitive events are supported relating to physical time (i.e. time events), to occurrences within databases (i.e. data manipulation events and transaction events), and to occurrences within application programs (i.e. abstract events). Composite events are made up of primitive and/or other composite events and event operators. The event operators conjunction, disjunction, sequence, iteration, and negation are applicable to events at local and at remote sites, whereas the concurrency event operator is specific to events at remote sites. This chapter included a discussion of event parameters. In particular, each event has a timestamp parameter indicating when and where it occurred. Timestamps play a special role in establising event semantics.

Chapter 6 addressed the semantics of events. The semantics identifies when and where an event occurs. Since the semantics of a composite event depends on the timestamps of constituent events, the structure and handling of timestamps must take account of the notions of physical time and temporal order in distributed systems. It was essential to determine the temporal relationship between timestamps; that is, whether two timestamps denote concurrent or sequential events. First, the general semantic model was developed. Second, it was recognised that the fulfillment of a simple assumption, namely if there are no two primitive event occurrences at the same site within one global clock tick, leads to a simplified semantic model. This assumption is satisfied when the frequency of primitive event occurrences is sufficiently low. Finally, the semantics of composite events was formally defined for both the general and the simplified semantic model.

Event detection was discussed in Chapters 7 and 8. Chapter 7 identified the goals of event detection, defined the system architecture, and described the basic detection mechanisms: global event trees representing global composite events are evaluated concurrently at arbitrary sites of a distributed system. Chapter 8 presented the algorithms for evaluating the nodes of global event trees. Two evaluation policies were considered: asynchronous and synchronous evaluation. Differences between the two are due to the different handling of site failures and network delays. In asynchronous evaluation, nodes are evaluated irrespective of site failures and network delays, whereas these are taken account of in synchronous evaluation.

A prototype has been implemented realising the detection of global composite events with both asynchronous and synchronous evaluation. Timestamps have been implemented with respect to the simplified semantic model. Chapter 9 outlined the system architecture, the data structures, the system setup and initialisation, and the system's runtime behaviour. Further, several tests were described which illustrate the differences between asynchronous and synchronous evaluation.

10.2 Outlook

Further work can be divided into the following tasks:

- *Optimisation:* In a distributed system, it is essential to minimise event traffic, that is, to minimise the network load which is due to event monitoring. One possibility is to realise event detection with network object pointers (see Section 9.6.4). This saves copying of event data, i.e. of event parameters. Another possibility is event filtering in order to send only relevant event occurrences to remote event detectors. In this dissertation event filtering was achieved using parameter restrictions (see Section 5.3.3). A more general approach merging events and conditions is needed.

- *Design:* Specifying a desired system behaviour is a difficult task since event types are scattered around the distributed system. Before specifying a composite event type, relevant constituent event types have to be located. Moreover, the detector site of a global composite event type has to be decided. In order to locate relevant event types, each site has to be extended with an event interface expressed in some common interface definition language (IDL)[1]. In order to decide on a detector site, some heuristic algorithm has to be applied, either by the user specifying the event type, or by the system.

- *Supporting system evolution:* Taking up the previous point, system evolution has to be considered. Event detectors need to be adaptable to system changes; that is, new event types need to be implemented and existing event types need to be re-implemented.

- *Verifying runtime behaviour:* Detected event occurrences participate in the detection of other event occurrences. Hence, the overall effects of event monitoring are very complex. It may be convenient to use concurrency theory, i.e. CCS (Calculus for Concurrent Systems) [**?**] or one of its extensions, to formally verify runtime behaviour.

- *Extending distributed object system standards:* COM (Component Object Model) [**?**] and CORBA (Common Object Request Broker Architecture) [**?**] provide basic event services. It would be interesting to investigate how these standards need to be extended in order to perform system monitoring as described in this dissertation.

- *Security:* It is necessary to secure event monitoring, that is, to provide access control to event detection mechanisms. Moreover, since detected events might relate to the internal behaviour of objects, there is a tension between object encapsulation and event detection. The security issue needs to be addressed before event-driven monitoring can be employed on a widespread basis.

[1]Note, that the underlying distributed system may be heterogeneous.

Appendix A

Event Traces

A.1 Test Two

A.1.1 Asynchronous Evaluation

The following output represents the first and third part of the "results of asynchronous evaluation" (see Figure **??**).

```
osprey.cl.cam.ac.uk 1 :
     Time of occurrence:  1996 Mar 6 16:22:7:797915 ticks 589277
kookaburra.cl.cam.ac.uk 1 :
     Time of occurrence:  1996 Mar 6 16:22:7:754039 ticks 589277
pelican.cl.cam.ac.uk 1 :
     Time of occurrence:  1996 Mar 6 16:22:7:730449 ticks 589277
osprey.cl.cam.ac.uk 2 :
     Time of occurrence:  1996 Mar 6 16:22:9:890046 ticks 589298
pelican.cl.cam.ac.uk 2 :
     Time of occurrence:  1996 Mar 6 16:22:12:893662 ticks 589328
Event Detected :  osprey.cl.cam.ac.uk 10
     Time of occurrence:  1996 Mar 6 16:22:9:890046 ticks 589298
       No interval
       osprey.cl.cam.ac.uk Lower bound
     Time of detection:  1996 Mar 6 16:22:9:948645 ticks 589299
       No interval
       osprey.cl.cam.ac.uk Lower bound
Event Detected :  kookaburra.cl.cam.ac.uk 10
     Time of occurrence:  1996 Mar 6 16:22:9:890046 ticks 589298
       No interval
       osprey.cl.cam.ac.uk Lower bound
     Time of detection:  1996 Mar 6 16:22:12:474705 ticks 589324
       No interval
       kookaburra.cl.cam.ac.uk Lower bound
Event Detected :  pelican.cl.cam.ac.uk 10
     Time of occurrence:  1996 Mar 6 16:22:12:893662 ticks 589328
       No interval
       pelican.cl.cam.ac.uk Lower bound
     Time of detection:  1996 Mar 6 16:22:12:937007 ticks 589329
       No interval
       pelican.cl.cam.ac.uk Lower bound

osprey.cl.cam.ac.uk 1 :
     Time of occurrence:  1996 Mar 6 16:51:54:193092 ticks 607141
kookaburra.cl.cam.ac.uk 1 :
     Time of occurrence:  1996 Mar 6 16:51:54:168841 ticks 607141
pelican.cl.cam.ac.uk 1 :
     Time of occurrence:  1996 Mar 6 16:51:54:136503 ticks 607141
osprey.cl.cam.ac.uk 2 :
     Time of occurrence:  1996 Mar 6 16:51:56:306361 ticks 607163
pelican.cl.cam.ac.uk 2 :
     Time of occurrence:  1996 Mar 6 16:51:59:214155 ticks 607192
Event Detected :  osprey.cl.cam.ac.uk 10
     Time of occurrence:  1996 Mar 6 16:51:56:306361 ticks 607163
       No interval
       osprey.cl.cam.ac.uk Lower bound
     Time of detection:  1996 Mar 6 16:51:56:373105 ticks 607163
       No interval
       osprey.cl.cam.ac.uk Lower bound
Event Detected :  kookaburra.cl.cam.ac.uk 10
     Time of occurrence:  1996 Mar 6 16:51:56:306361 ticks 607163
       No interval
       osprey.cl.cam.ac.uk Lower bound
     Time of detection:  1996 Mar 6 16:51:56:589191 ticks 607165
       No interval
       kookaburra.cl.cam.ac.uk Lower bound
```

A.1.2 Synchronous Evaluation

The following output represents the first part of the “results of synchronous evaluation” (see Figure **??**).

```
osprey.cl.cam.ac.uk 1 :
     Time of occurrence:  1996 Mar 6 16:29:47:909522 ticks 593879
kookaburra.cl.cam.ac.uk 1 :
     Time of occurrence:  1996 Mar 6 16:29:47:906172 ticks 593879
pelican.cl.cam.ac.uk 1 :
     Time of occurrence:  1996 Mar 6 16:29:47:905278 ticks 593879
osprey.cl.cam.ac.uk 2 :
     Time of occurrence:  1996 Mar 6 16:29:49:998521 ticks 593899
pelican.cl.cam.ac.uk 2 :
     Time of occurrence:  1996 Mar 6 16:29:53:6331 ticks 593930
Event Detected :  osprey.cl.cam.ac.uk 10
     Time of occurrence:  1996 Mar 6 16:29:49:998521 ticks 593899
       No interval
       osprey.cl.cam.ac.uk Lower bound
     Time of detection:  1996 Mar 6 16:29:50:55478 ticks 593900
       No interval
       osprey.cl.cam.ac.uk Lower bound
Event Detected :  kookaburra.cl.cam.ac.uk 10
     Time of occurrence:  1996 Mar 6 16:29:49:998521 ticks 593899
       No interval
       osprey.cl.cam.ac.uk Lower bound
     Time of detection:  1996 Mar 6 16:29:52:186849 ticks 593921
       No interval
       kookaburra.cl.cam.ac.uk Lower bound
```

A.2 Test Three

A.2.1 Asynchronous Evaluation

The following output represents the "results of asynchronous evaluation" (see Figure **??**).

```
pelican.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 12:4:2:492346 ticks 434424
kookaburra.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 12:4:2:96069 ticks 434429
kookaburra.cl.cam.ac.uk 2 :
      Time of occurrence:  1996 Mar 7 12:4:3:146204 ticks 434431
kookaburra.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 12:4:4:22635 ticks 434442
kookaburra.cl.cam.ac.uk 2 :
      Time of occurrence:  1996 Mar 7 12:4:5:296489 ticks 434452
kookaburra.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 12:4:6:37663 ticks 434463
kookaburra.cl.cam.ac.uk 2 :
      Time of occurrence:  1996 Mar 7 12:4:7:456776 ticks 434474
pelican.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 12:4:8:578668 ticks 434485
Event Detected :  pelican.cl.cam.ac.uk 10
```

```
      Time of occurrence:  1996 Mar 7 12:4:2:492346 ticks 434424
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 12:4:2:632943 ticks 434426
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Number of left child events 0
Event Detected :  kookaburra.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 12:4:3:146204 ticks 434431
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 12:4:3:306235 ticks 434433
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
Event Detected :  kookaburra.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 12:4:5:296489 ticks 434452
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 12:4:5:646549 ticks 434456
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
Event Detected :  kookaburra.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 12:4:7:456776 ticks 434474
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 12:4:7:586801 ticks 434475
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
Event Detected :  pelican.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 12:4:8:578668 ticks 434485
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 12:4:8:623445 ticks 434486
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Number of left child events 2

pelican.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 13:44:28:997261 ticks 494689
kookaburra.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 13:44:29:155093 ticks 494691
kookaburra.cl.cam.ac.uk 2 :
      Time of occurrence:  1996 Mar 7 13:44:30:245246 ticks 494702
```

```
kookaburra.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 13:44:31:325397 ticks 494713
kookaburra.cl.cam.ac.uk 2 :
      Time of occurrence:  1996 Mar 7 13:44:32:41555 ticks 494724
kookaburra.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 13:44:33:785699 ticks 494737
kookaburra.cl.cam.ac.uk 2 :
      Time of occurrence:  1996 Mar 7 13:44:34:86585 ticks 494748
pelican.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 13:44:35:35785 ticks 494750
Event Detected :  pelican.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 13:44:28:997261 ticks 494689
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 13:44:29:95136 ticks 494690
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Number of left child events 0
Event Detected :  kookaburra.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 13:44:30:245246 ticks 494702
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 13:44:30:375272 ticks 494703
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
Event Detected :  kookaburra.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 13:44:32:41555 ticks 494724
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 13:44:33:11563 ticks 494731
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
Event Detected :  kookaburra.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 13:44:34:86585 ticks 494748
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 13:44:35:115901 ticks 494751
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
Event Detected :  pelican.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 13:44:35:35785 ticks 494750
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 13:44:35:96098 ticks 494750
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Number of left child events 1

pelican.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 14:2:12:85946 ticks 505328
kookaburra.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 14:2:13:1087 ticks 505330
kookaburra.cl.cam.ac.uk 2 :
      Time of occurrence:  1996 Mar 7 14:2:14:71226 ticks 505340
kookaburra.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 14:2:15:131368 ticks 505351
kookaburra.cl.cam.ac.uk 2 :
      Time of occurrence:  1996 Mar 7 14:2:16:201512 ticks 505362
kookaburra.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 14:2:17:311659 ticks 505373
kookaburra.cl.cam.ac.uk 2 :
      Time of occurrence:  1996 Mar 7 14:2:18:391805 ticks 505383
pelican.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 14:2:18:9754 ticks 505389
Event Detected :  pelican.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 14:2:12:85946 ticks 505328
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 14:2:13:33516 ticks 505330
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Number of left child events 0
Event Detected :  kookaburra.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 14:2:14:71226 ticks 505340
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 14:2:14:171248 ticks 505341
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
Event Detected :  kookaburra.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 14:2:16:201512 ticks 505362
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 14:2:16:471566 ticks 505364
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
```

```
Event Detected :  kookaburra.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 14:2:18:391805 ticks 505383
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 14:2:18:641845 ticks 505386
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
Event Detected :  pelican.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 14:2:18:9754 ticks 505389
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 14:2:19:45371 ticks 505390
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Number of left child events 3
```

A.2.2 Synchronous Evaluation

The following output represents the first part of the "results of synchronous evaluation" (see Figure ??).

```
pelican.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 11:48:53:259937 ticks 425332
kookaburra.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 11:48:53:410016 ticks 425334
kookaburra.cl.cam.ac.uk 2 :
      Time of occurrence:  1996 Mar 7 11:48:54:489702 ticks 425344
kookaburra.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 11:48:55:569388 ticks 425355
kookaburra.cl.cam.ac.uk 2 :
      Time of occurrence:  1996 Mar 7 11:48:56:689072 ticks 425366
kookaburra.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 11:48:57:788752 ticks 425377
kookaburra.cl.cam.ac.uk 2 :
      Time of occurrence:  1996 Mar 7 11:48:58:878435 ticks 425388
pelican.cl.cam.ac.uk 1 :
      Time of occurrence:  1996 Mar 7 11:48:59:359274 ticks 425393
Event Detected :  pelican.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 11:48:53:259937 ticks 425332
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 11:48:53:583094 ticks 425335
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Number of left child events 0
Event Detected :  kookaburra.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 11:48:54:489702 ticks 425344
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 11:48:154:599669 ticks 425345
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
Event Detected :  kookaburra.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 11:48:56:689072 ticks 425366
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 11:48:56:90901 ticks 425369
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
Event Detected :  kookaburra.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 11:48:58:878435 ticks 425388
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 11:48:59:118363 ticks 425391
        No interval
        kookaburra.cl.cam.ac.uk Lower bound
Event Detected :  pelican.cl.cam.ac.uk 10
      Time of occurrence:  1996 Mar 7 11:48:59:359274 ticks 425393
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Time of detection:  1996 Mar 7 11:49:5:986357 ticks 425459
        No interval
        pelican.cl.cam.ac.uk Lower bound
      Number of left child events 3
```

Bibliography

[Bac92] J. Bacon. *Concurrent Systems*. Addison-Wesley Publishing Company, 1992.

[BNOW94] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. Technical Report 115, Systems Research Center, Digital Equipment Corp., 1994.

[Har92] S.P. Harbison. *Modula-3*. Prentice Hall, 1992.