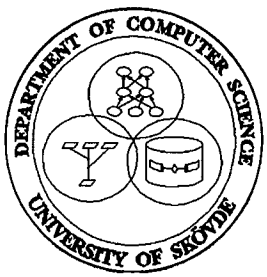

UNIVERSITY OF SKÖVDE
Department of Computer Science

Jonas Mellin (jonas.mellin@ida.his.se)

Technical Report No HS-IDA-TR-96-005

Event Monitoring & Detection in Distributed Real-Time Systems: A Survey

Jonas Mellin

June 1996

HS-IDA-TR-96-005

Event Monitoring & Detection in Distributed Real-Time Systems: A Survey *

Jonas Mellin
Department of Computer Science,
University of Skövde
Box 408, S-541 28 Skövde, Sweden
jonas.mellin@ida.his.se

June 7, 1996

Abstract

This report is a survey of monitoring and event detection in distributed fault-tolerant real-time systems, as used in primarily active database systems, for testing and debugging purposes. It contains a brief overview of monitoring in general, with examples of how software systems can be instrumented in a distributed environment, and of the active database area with additional constraints of real-time discussed. The main part is a survey of event monitoring mostly taken from the active database area with additional discussion concerning distribution and fault-tolerance. Similarities between testing and debugging distributed real-time systems are described.
keywords: event monitoring, distribution, real-time, active databases, testing, debugging, software engineering

*This work was supported by NUTEK(The Swedish National Board for Industrial and Technical Development), as part of the Distributed Reconfigurable Real-Time Database Systems Project in the Embedded Systems Program.

Contents

List of Figures

List of Tables

1 Introduction

Richard Snodgrass starts off his article [Sno88] with the following definition:

> "Monitoring is the extraction of dynamic information concerning a computational process, as that process executes. This definition encompasses aspects of observation, measurement, and testing. Much has been written about monitoring uniprocessor systems, and the general techniques for tracing and sampling are well established. These approaches do not scale well to monitoring complex systems, which include large uniprocessors, tightly coupled multiprocessor systems, and [distributed systems]."

Monitoring becomes an interesting issue in complex systems as the problem of observing a complex system from within the system will cause an intrusion in the system and affect the system behavior. This is referred to as the *probe-effect* (see section 2.2) and is important to avoid in real-time systems. For example, if testing of a real-time system requires introduction of extra code and hardware, causing a probe-effect during testing, this extra code and hardware cannot be removed in the operational system because the system would behave differently and effectively invalidate many of the tests. In non real-time systems where there are no deadlines, the extra code and hardware can be removed as long as it does not affect the logical behavior of the system. Proper monitoring for testing must thus be considered in the design of a real-time system.

Complex real-time systems need to monitor tasks to be able to schedule them so that they will meet their dead-lines. Fault-tolerant systems need monitoring to observe whether or not a component fails so that the system can handle the failure without exhibiting a failure outside the system. All safety-critical real-time systems must include fault-tolerance. In (re)active database systems, monitoring of events is mandatory and needed for handling rules in an efficient manner. Software tools for testing and debugging need to be able to observe the state of the system or continuous changes in the system.

1.1 Basic Definitions

Basic definitions concerning real-time systems, fault-tolerance, distribution, and active databases are presented as a brief guide the pinpointing the background of this survey.

1.1.1 Real-time system

The need for real-time systems is increasing because some applications — e.g., safety-critical applications such as aircraft control and nuclear plant control, and non-safety-critical systems such as multimedia applications — need computer systems that are not only logically correct but will produce responses in a timely (and predictable) manner. There are several definitions of real-time systems such as [KV94]:

> "The computer system must react to stimuli from the controlled object (or the operator) within time intervals dictated by its environment."

or [You82]:

> "any information processing activity or system which has to respond to externally-generated input stimuli within a finite and specified period"

In [BW89] the latter definition is considered to be too broad and systems are divided into *hard* and *soft* real-time systems. In a *hard* real-time system it is absolutely imperative that a task finishes before its deadline. In a *soft* real-time system a missed deadline will not result in a catastrophic failure and it may be useful to complete the task even if the deadline was missed. In

addition a *firm* real-time system can be considered as a special case of a soft real-time system where there is no value in executing a task has no value after its deadline.

Best-effort systems are normally thought of as soft real-time systems, but they are more correctly placed in between *hard* and *soft* systems. A *best-effort* real-time system will be able to handle all hard deadlines and most of the soft deadlines during normal load. During peak load the system will degrade gracefully and only the hard deadlines are guaranteed [KV94].

Further classification of *hard* real-time systems is *fail-safe* vs *fail-operational* [KV94]. In a *fail-safe* system one or more safe states can be accessed in case of a system failure. Applications in which such safe states cannot be identified must be *fail-operational*, i.e., they have to provide a minimum level of service even in the case of faults occurring.

Real-time systems are further divided into *time-triggered systems* — i.e., systems which execute in lock step and react to external events at prespecified instants, and *event-triggered systems* — i.e., systems which react to significant external events immediately [KV94].

Event Showers Event triggered real-time systems are prone to *event showers* [KV94, p. 426, p. 456]. These may occur in situations of exception (fire, leakage, explosion, crash, etc), where a number of nodes provide alarm information, some of it redundant or repeated, some of it multiplied by propagation, although often concerning a single cause. In a "pure" event triggered system, where no events are transformed nor constrained before they are passed into the system, this may cause an overload in the system.

No system can handle a completely random load. *Aperiodic* events, i.e., events whose interarrival time is undetermined, are not tractable deterministically. *Sporadic* events are a way of constraining the environment so that the events will have a minimum interarrival time, i.e., they are tractable [KV94].

Furthermore, the subsequent event generation can be predicted to a significant extent when an alarm occurs, i.e, most events following a first alarm are predictable rather than chance events. Rules can be created by the designer [KV94]: i) to *compact* successive instantiations (repeated events) of the same alarm at the *representatives* — i.e., an internal entity that acts on or observes a real-world object [KV94]; ii) to *discard* redundant events from different sources, either at the representatives or upon arrival at the node; iii) to *prepare* communication and computing resources for the forthcoming shower.

Additional engineering measures can further improve the event handling capability. These have to do with load or flow control, concerned with regulating the flow of data from periphery (representatives) to the nucleus of the system (node). For example, if the real world events are bursty the transmission from the representative to the nodes can be *smoothed*, by spacing them by the equivalent of an average rate [KV94] — also known as transforming aperiodic events into sporadic events.

1.1.2 Fault Tolerance

Two important hypotheses must be postulated about the environment since a computer system has a finite amount of resources in terms of processing capacity, available memory, I/O ports etc, all of which can fail. These hypotheses are the *Load Hypothesis* and *Fault Hypothesis* [KV94].

The *Load Hypothesis* characterizes the *peak load* in terms of maximum rate of — or minimum time interval between — events. Hard and best-effort real-time systems must be designed to handle peak loads.

The *Fault Hypothesis* characterizes the types

and frequency of anticipated faults. In a worst case scenario in a dependable hard or best-effort real-time system, the system must be able to handle peak load at the same time as the maximum number of faults occur. An example of a common part of the fault hypothesis is the *single failure assumption*, i.e., no error will occur while trying to handle another error — including recovering from the error, repairing and reintegrating the failed component.

As hardware occasionally fails to operate, and design faults are impossible to avoid, fault-tolerant software has to be built. However, noone has so far been able to find, nor is it likely that anyone will find, a software engineering method for the development of complex computer systems which will guarantee that design faults are not introduced. A fault-tolerant system needs redundancy in space and time to be able to handle faults. In software this means the addition of extra code.

There are two strategies for making software fault-tolerant. One approach is *active fault-tolerance* [KV94] — the extra fault-tolerant code is always executing whether or not errors occur. It is also known as *static fault-tolerance* [BW89] — the behavior of the software does not change because an error occurs. The main technique is called *N-modular redundancy* [BW89, KV94]. The basic idea behind N-modular redundancy is that separate modules are doing the same work and the results are compared by a voter [BW89]. If m modules are allowed to fail then $m+1$ modules are needed if they exhibit the *fail-silent* property, i.e., the voter will not receive result from any of the faulty modules. Further, $2m+1$ modules are needed to mask the m arbitrary failures, i.e., the voter takes the majority. Finally, $3m+1$ modules are needed to be able to detect which modules failed.

The other approach is *passive fault-tolerance* [KV94] (as opposed to active), because the error handling is only invoked whenever an error occurs. It is also known as *dynamic fault-tolerance* [BW89] (as opposed to static), because the behavior of the software changes when an error occurs, The main techniques are *backward* and *forward error recovery* in which the system recovers from an error when it is detected either by rolling back to a safe (saved) state or by rolling forward to an anticipated safe state.

The use of active — or static — fault-tolerance is the preferred method in hard real-time systems because it does not require any extra time to handle an error [KV94]. Instead extra hardware and software is required. Passive — or dynamic — fault-tolerance will always cost extra time which must be included in the worst-case scenario when scheduling tasks. The major problem with passive fault-tolerance is that this extra cost in time may make scheduling too pessimistic and result in unnecessarily low resource usage (unless some tasks may be discarded as in a soft real-time system).

In passive fault-tolerance the need for monitoring is intuitive, i.e., if an error occurs then perform recovery. For example, if an actuator is connected to fail-silent modules that will tell the system that a fault occurred, a hot standby module can take over the faulty modules' roles [KV94]. In this case monitoring can be used for detecting the event that a module has failed and determine that the hot standby is supposed to take over.

1.1.3 Distribution

In this report a distributed system is viewed as loosely coupled processing elements (nodes) that do not share memory nor a global clock. A distributed system can be either homogeneous or heterogeneous. In a homogeneous system all nodes share the same data representation and data alignment and normally execute the same operating system. Only homogeneous systems will be discussed in this survey.

A distributed system is potentially fault-tolerant. This might seem to be a contradiction because there are even more components that can fail in a distributed system than in a conventional system. However, a distributed system does not necessarily fail when the maximum number of nodes allowed by the fault hypothesis fail independently, because the nodes can be replicated. Furthermore a node can be made fail-silent by replicating and comparing constituent components on the node.

A distributed real-time system must be supported by reliable real-time communication in order to guarantee response time of real-time remote requests. The properties *tightness*, *bounded transmission delay*, *bounded omission degree* and *bounded inaccessibility* are necessary for real-time communication [Ver94b]. These properties support: i) enforcing a bounded delay from request to delivery of a frame, given the worst case load conditions assumed; ii) ensuring that a message is delivered despite the occurrence of omissions; and iii) maintaining connectivity.

In order to enforce bounded transmission delay of a message, issues such as *traffic patterns* (when and with which frequency requests are made), *latency classes* (a message in the lowest latency class is of the highest urgency), LAN sizing and parameterizing, and user-level load/flow control, must be taken into account. The fault hypothesis must include assumptions about omission failures concerning the network which will lead to protocols that ensure that messages are delivered even though an omission fault occurred. Fortunately omission failures are rare and of a bursty nature, i.e., it is normally safe to assume that only one omission will occur during delivery of a message [Ver94b].

The handling of CPU (and process) groups — e.g., multicast — are a central idea in reliable real-time communication which, for an efficient implementation, pervades all layers in the network protocol. The main reasons for the importance of group protocols can be found in [Ver94b, p 479]:

> "Real-life experience has shown that [the group protocols] drastically simplifies the algorithmic and correctness problems of distributed computing"

Global Time Base and Global Ordering: Global ordering of events and requests must be preserved in a distributed system in order to prove global properties about the system. Without global ordering it is impossible to ensure that requests are served in the order they were made, nor is it possible to tell in which order two (causally dependent) events occurred. In a distributed real-time system, a global timebase must also be supported in otherwise it is impossible to i) synchronize the triggering of actions at two different nodes, ii) do distributed logging, and hence global event detection, and iii) support *replica determinism* — i.e., letting independent replicated modules come to the same result when they are executing algorithms that are dependent on time, order, and priority of requests (and events) [KV94].

The $\delta_t - precedence$ relation, i.e., δ_t is the minimum real time interval for causal relation to be generated [Ver94a], is the criterion for potential causality. The δ_t is $2g$ given a clock granularity g [Ver94a], which must be at least equivalent to the bounded clock precision π (cf clock-synchronization algorithms [Chr89, CGG94]). However, choosing a too small g will lead to a too dense time base where too many undesirable repeated observations may be performed [Ver94a], e.g., too many repeated occurrences during event shower may be let through from the representatives.

Ordering by logical clocks [Lam78] can be used in non-real-time distributed system, however,

this will not work properly if any participants (e.g., processes, nodes) exchange messages outside the ordering protocol [Ver94a, 474]. Furthermore, it has been shown that it is impossible to order a sequence of three causally independent events (a.k.a *strong gap detection*[BR93]) timestamped with logical clocks using this protocol. However, it is possible to use *vectorized logical clocks*, where each time-stamp contains a logical clock for each node in the system, to detect that an event E_1 did not occur before E_2 which occurred before E_3 (a.k.a *weak gap detection* [BR93]).

1.1.4 Active Databases

Active databases is an active research area where the early work was presented in the context of HiPAC [DBB+88], ETM [KDM88] and POSTGRES [SHH87]. These can be classified as event-triggered systems (not necessarily real-time systems). The major idea is to add reactive mechanisms to the database as ECA-rules. An ECA-rule is an association of an Event, a Condition and an Action. When an event occurs the corresponding rules are triggered and their conditions evaluated. For all rules that have been triggered, if their conditions are true their associated actions are performed. Consensus on what events are, how events should be detected, how the rules should be executed has only been partially reached. There are still open research issues in this area.

However, active databases are in need of event monitoring and event detection. It is our contention that this event monitoring and detection can be combined with the monitoring and detection needed for other components such as software engineering tools and real-time applications.

Coupling Modes: Event monitoring (specification part) in active databases are affected by the *coupling modes* (i.e., modes describing when and how the handling of the sequence event monitoring, condition evaluation, and action execution of a rule takes place [DBB+88, Buc94]), which is described in section 3. For example, the condition can take place immediately after the event has been detected, whereas, the action can be executed in a separate (detached) transaction.

1.2 Basic Issues

Below basic issues in software engineering are discussed.

1.2.1 Software Engineering Issues

Testing and debugging are two important activities in any software engineering methodology. Debugging is often not explicitly mentioned in software engineering methodologies with the exception of methodologies for building expert systems and other systems based on artificial intelligence (cf RUDE [Par86]). Testing, however, is seen as an important activity in all methodologies.

The area of software testing has been an active area of research since the 1960s. Practical experience, empirical and theoretical studies have led to well-defined methodologies for testing software on different levels [Bei90]. However, this has not solved the problem of proving in a reasonable time how dependable a system is nor does there exist any good measure of software quality. According to [Bei90] all simple faults are caught with testing but the subtle interaction faults between components are left in the system.

Even if this pessimistic view of testing is taken, testing is a necessary step in software engineering used in conjunction with well known methodologies for specification and controlling the software development process. When testing distributed real-time systems the major problem in executing the tests are observability and re-

producibility [Sch94, Sch95a].

As mentioned, when a system is observed there is an intrusion known as the probe-effect (see section 2.2), because *software sensors* in the form of extra code need to be added. In software, this intrusion will affect the temporal behavior of the system. This is particularly severe for real-time systems. The probe-effect must be avoided by *leaving* the software sensors in the operational system [Sch94, Sch95a].

It must be possible to re-run a test and it should produce the same result every time unless the system is changed. In order to do this, recorded time is required, i.e., significant events must occur at the same points in time during replay as in the original execution and, hence, the occurrences of significant events must recorded and replayed [Sch94, Sch95a]. Furthermore, deviations in the hardware must be hidden. Care must be taken so that additional (or different) probe-effect is not introduced when the system is observed under a re-run of a test.

Furthermore, testing must be considered already in the specification and design phase. It cannot be delayed until after the design, especially in a real-time system since the tests may change the temporal behavior of the system.

1.3 Overview

In the remainder of this survey we consider Monitoring in section 2, Event Monitoring in section 3, Debugging in section 4, and Conclusion in section 5.

2 Monitoring

The term *monitor* is used for various constructs with different meanings. In synchronization a monitor is an abstract data type providing the programmer with an easy way of synchronizing accesses of a resource of some kind [Hoa74, And91] and the term has also been used earlier as a synonym for operating systems [SG94]. The basic semantics of this kind of monitors are of control rather than observation.

In other work the monitor is a construct for monitoring of events or properties. Monitoring in this case is the extraction of dynamic information concerning the computational process [Sno88]: for example, the violation of timing constraints [JRR94] and events in real-time systems [Pla84]. This survey concentrates on this (in the same para.) latter meaning of the concept of a monitor.

The traditional steps (in italic) in monitoring computer systems occur during the following four phases:

1. Design and implementation of a system: a) *Sensor configuration*: This step involves deciding what information each sensor will record (and possibly report) and where the sensor will be located; b) *Sensor installation*: The sensors must be coded (or built in hardware) and inserted in the correct location in the subject system. Provision must be made for temporary and permanent storage of the collected data in the sensor.

2. System setup: *Enabling sensors*: Some sensors are permanently enabled, storing (and possibly reporting) monitoring data whenever executed, while others may be individually or collectively enabled, directly or indirectly by directives from the user.

3. Execution of a system: *Data generation*: The subject program is executed, and the collected data are stored in main memory or on secondary storage for consumption now or later. No dynamic reconfiguration is generally possible during this phase, e.g., it is not possible to change what the sensors should record during run-time unless the sensors are designed to adapt to anticipated changes.

4. Post-execution analysis of the generated data: a) *Analysis specification*: In most system the user is given a menu of supported analyses; sometimes a simple command language is available. b) *Display specification*: The user is given a set of formats (either specified via a menu or a command language), ranging from a list of raw data packets to canned reports or simple graphics. c) *Data analysis*: Data analysis normally occurs in batch mode after the data have been collected due to the often high computational cost of the analysis. d) *Display generation*: Usually this step occurs immediately after data analysis, although a few monitoring packages allow the analyzed data to be displayed at a later time.

These steps were reduced in [Sno88], by using a declarative relational approach to specifying monitors to the following: 1a) Sensor configuration, 1b) Sensor installation, 4a) Analysis specification, 4b) Display specification and finally 3) Execution (where the missing steps of enabling sensors, data analysis, and display generation are performed automatically in the Execution step). Snodgrass uses *historical databases* and a relational query language *TQuel* as a basis this simplification. This approach makes it easier to maintain and extend compared to the traditional approach. The major drawbacks mentioned by Snodgrass are that:

1. the queries are specified before the data is collected. This a priori knowledge is not always available.

2. the complexity of the relational monitor makes it very inefficient. In a practical test between this approach in TQuel, including optimization of the queries, and a traditional procedural approach (in LISP) the difference in performance was two order of magnitudes.

The first drawback is not so severe in a real-time system compared to a conventional system as more a priori knowledge about the environment and the computer system is known.

The second drawback is quite severe in a real-time systems if the relational approach would be used for on-line analysis. The detection of an event must be considerable shorter than the remaining time to the deadline to allow a computation to finish in time. If the worst-case estimation of how long it will take to detect an event is too pessimistic, or the time it takes to detect an event is not negligible compared to the time it takes to perform the task invoked as a result of the event, or the deadline is very close relative to the associated event occurrence, then the complexity of the relational monitors is a critical issue.

The most severe problem with relational monitors is that the detection of complex event patterns uses the Cartesian product operator, which may be applied several times. Optimizations (or improvements) of the TQuel queries are available, but these only exist for a few of the possible expressions in TQuel.

The relational approach suggested by Snodgrass has a lot of advantages, but the disadvantage of complexity makes it virtually useless for event detection in real-time systems. However, by using less generic, but still powerful languages based on operator grammars specifically designed for dealing with events the event detection can be made more efficient (see section 3). These languages must be extended with "display specification" and "display generation" otherwise these steps must be performed manually.

2.1 Hardware, Software and Hybrid Monitors

In INCAS [HW90] they differentiate between *hardware monitors* implemented using hardware

only, *software monitors* using software only and *hybrid monitors* which are implemented using a combination of both. Hybrid monitors are considered good because they the combine the flexibility of software monitors with the efficiency of hardware monitors. A drawback of hybrid monitors compared to software monitors is the need to build special purpose hardware, but this can normally be kept to a minimum.

2.2 Probe-effect

A monitor will affect the behavior of an observed system — a monitored task will have longer (possibly unpredictable) response times. Furthermore, a minor issue is that monitored applications may be larger than their unmonitored counterparts.

The probe-effect consists of two parts: i) the unpredictability of the monitor, and ii) the the difference in the behavior of a monitored system and its unmonitored counterpart. The unpredictability of the monitor can be solved by differentiating between the data collection part, i.e., the sensor, which can designed to be predictable, and the data processing part of the monitor. The sensor is placed within the observed system, whereas, the data processing part is placed outside the observed system. Furthermore, hybrid and hardware monitors can be used to considerably reduce the overhead of both the sensor and the data processing part compared to a software monitor. A difference in the behavior of a monitored system and its unmonitored counterpart can only be avoided by leaving the monitors in the operational system, however, this is only imperative in real-time systems [Sch94, Sch95a].

2.3 Event Log

The functionality of a monitor is to detect that an event has occurred or that a property has changed. If complex event patterns are allowed then the monitor needs to keep an event log. The monitor checks the event log and if a complex event pattern is detected the required action is taken, e.g., by passing the data about the event occurrence to another module.

If the event log is stored in stable memory the log can be used in a similar way to a "black box" in an aircraft. If the system fails, the event logs can be examined to determine why it failed.

2.4 Instrumentation

Sensor configuration and installation together with installation of the monitoring facility is referred to as *instrumentation.* Below are three important examples of instrumentation, the first two characterized by the use of hybrid monitors and the second characterized by the problems of instrumenting a closed architecture.

2.4.1 INCAS Distributed Monitoring Facility

In the INCAS [HW90] project a hybrid monitor has been used. The sensors are inserted into the software as instructions that are visible on the main CPU data bus which is snooped by the hybrid monitor — a general purpose CPU called a Test and Measurement Processor (*TMP*) using special hardware for bus snooping and running monitoring software. A node in INCAS consists of one main CPU with one or more TMPs hooked on — making the hybrid monitor fault-tolerant. Furthermore, it is possible to let the TMPs use a separate network so that they can communicate without interfering with normal communication between the nodes.

TMPs only cause approx. 1 % overhead in the execution of the monitored application while detecting events occurring at a fast rate (up to 1300 events/second). This hybrid monitor has been used for run-time monitoring of tasks for deadline scheduling [JRR94].

2.4.2 Monitoring using Coprocessors

In the work by Gorlick [Gor91] a different hybrid monitor, based on the use of a coprocessor, causes less overhead than the the monitor in the INCAS project. His main reason for using a coprocessor is the fact that bus-snooping of embedded real-time systems will be made impossible as more hardware components are added to avoid usage of the bus, e.g., cache memories. Another reason is to be able to handle more events per time unit than traditional bus snooping because the coprocessor interface is generally faster than the data bus.

2.4.3 Instrumentation of Object-Oriented Database Management Systems

In an object-oriented DBMS, classes are made persistent either by inheriting from a persistent class hierarchy or by attaching objects at run-time to objects handling persistence. The DBMS operations (which are methods) and arbitrary user method and procedures are likely to be instrumented.

There are four ways of instrumenting an object-oriented DBMS using software sensors:

i) the compilation process can be altered — a difficult task because compilers vary, language specifications change, and erroneous inputs must be handled. However, both DBMS operations and user specified methods/procedures are handled;

ii) call-back hooks in the DBMS can be used, however, experience has shown that no closed architectures have all the appropriate hooks [BZBW95];

iii) extend each class in the class hierarchy with an instrumented class, but the instrumentation will not be fully transparent as the user specified classes must inherit from the extended classes. Furthermore, instrumented user-specified classes that does not naturally relate to the DBMS class hierarchy may unnecessarily have to be an extension to the extended class hierarchy, e.g., a naturally non-persistent class may have to be designed as persistent class if it must be instrumented in an extended DBMS class hierarchy consisting of persistent classes;

iv) the existing classes in the class hierarchy can be changed, however, this can only be done in a public architecture where the source code is available and only DBMS operations can be handled.

The more flexible scheme of changing the compilation is desirable as it was concluded in [BZBW95] that extending the class hierarchy is relatively simple, but experience has shown that it often has the undesirable effect that all classes are persistent and all methods are instrumented.

3 Event Monitoring

A vital part of monitoring is to detect events when they occur and pass them to other modules. In an active database, a condition is evaluated (which may partly be performed by the monitor) if the associated event has been detected and an action is taken if the evaluation of the condition is true.

At the conceptual level there is an *event specification language* in which the user can express what kinds of events and event patterns that are interesting. This specification is translated into some internal representation, such as finite state automata in ODE [GJS93], Petri-nets in SAMOS [GD93], or *event graphs* in Snoop [CM93, CKAK93]) (an event graph is based on the syntactic trees that are produced for pars-

ing operator grammars [AU77], a.k.a extended syntactic trees [Deu94]). The internal representation is used to control the actual event monitoring.

Below event specification (section 3.1), event detection and composition (section 3.2), and expressibility vs efficiency and predictability are presented.

In this section important definitions from ODE, Snoop, SAMOS and REACH are discussed and evaluated. For each concept, a suggested "best" definition is presented.

3.1 Event Specification

3.1.1 Basic Event Definitions

Event: To the author's knowledge the clearest definition of an *event* is that it is instantaneous and atomic, i.e., either it happens completely or not at all (in Snoop [CM93, CKAK93]).

In the SAMOS prototype an event is said to indicate a point in time when the DBMS has to react [GD93], but in a real-time system there are also components that need to be able to react on events. The reason for this is that real-time systems are by their "nature" reactive systems but do not always require a DBMS. It is possible to argue that everything should be centered around a DBMS, but this is undesirable in cases where the functionality of the DBMS is not needed in all or part of the application. Hence the definition is too restrictive for real-time systems. The fact that an event is a point in time is the most important fact in the SAMOS definition.

In the REACH prototype an event is simply seen as the activator of a rule [Deu94], without saying anything about the nature of an event. However, it is explicitly stated that the set of events must comprise arbitrary method invocation events, temporal events and flow-control events in an OODBMS [Buc94, BZBW95].

In ODE an event is referred to as a "single activity" [GJM93], "happening of interest" [JMS92] and "an event known and supported by the database system" [GJ92a]. None of these definitions are formal nor are any properties clearly expressed.

Non-instantaneous events are considered by researchers in mathematics and logics [Gal9x], however, to the author's knowledge no evidence has been found so far to invalidate an instantaneous event model in active databases, real-time systems and software engineering tools.

Definition 1 *An* **event** *is atomic and instantaneous.*

Interval: Unlike an event an *interval* can stretch over a period of time. This definition is needed to explain how certain non-instantaneous phenomena are monitored. A point in time is normally viewed as a special case of an interval [CM93].

Definition 2 *An* **interval** *can stretch over a period of time, where an event is considered to be a special case of an interval.*

Definite Event: In Snoop [CM93] a *definite event* is an event that is guaranteed to occur by definition, e.g., a periodical temporal recurring event will always occur. In order to be able to guarantee that definite events can exist in a system some strategy for fault-tolerance must be considered — especially in distributed systems where hardware and software components may fail independently. For example, the data about an event occurrence can be lost or never occur (omission fault) or a faulty and malicious component may generate arbitrary events (Byzantine fault). Even in this case only a number of failures of prespecified types can be handled, but it is possible to state under which circumstances an event is guaranteed, e.g., a distributed system can be designed so that it tolerates up to k omission failures on the network.

Definition 3 *A* **definite event** *is an event that is guaranteed to occur, even in the presence of a prespecified maximum number of failures of certain prespecified failure types.*

Causal Precedence: In Snoop [CM93] an event (e_1) *causally precedes* another event (e_2) if and only if the occurrence of e_1 causes the occurrence of e_2. As in the case of the definite event the system must be designed to handle failures so that it is possible to guarantee causal precedence to a certain degree otherwise an event e_1 does not guarantee that an event e_2 will follow.

Definition 4 *An event (e_1)* **causally precedes** *another event (e_2) if and only if the occurrence of e_1 causes the occurrence of e_2, even in the presence of a prespecified maximum number of failures of certain prespecified failure types.*

Logical and Physical Event: In Snoop [CM93] a *logical event* corresponds to the specification of an event at the conceptual level and does not specify the *physical event* — e.g., machine instruction, timer interrupt — to which it will be mapped. Logical events are mapped to a physical/internal event either using a mapping specification — i.e., transparent instrumentation is done automatically by the compiler — or by choosing an implementation that corresponds to the mapping specification — i.e., non-transparent instrumentation done manually.

This makes a clear distinction between the issue of specification and implementation (instrumentation). Furthermore it clarifies how intervals — e.g., method calls, transactions — can be mapped into logical events such as beginning of interval and end of interval, e.g., beginning of transaction.

In the ODE prototype [GJ92b, JMS92, GJM93] the definition of a "logical event" is an event that is associated with a condition where the event occurs if and only if the physical event to which the event is mapped occurs and the associated condition holds true. This is similar to the use of generalized predicate path expressions for detecting events [And79, BH83]. This definition is not used in this survey.

In this survey a *filtered event* is a logical event (as defined in the Snoop prototype) associated with a filtering predicate.

Definition 5 *A* **logical event** *is the representation of an event on the conceptual level.*

Definition 6 *A* **physical/internal event**, *to which a logical event is mapped, is the actual event that occurs when the logical event is said to occur.*

Definition 7 *A* **filtered event** *is an event that occurs if and only if an associated condition holds true.*

All events discussed henceforth in this survey are logical events unless otherwise specified.

Simultaneous Events: In the Snoop prototype [CM93, CKAK93] simultaneous event occurrences are discussed, but to simplify the problem no two events are assumed to actually occur simultaneously.

In the ODE platform [GJ92a, JMS92, GJM93] each event occurrence is assumed to have a unique tuple with an event identifier that is monotonically increasing as events occur (and time passes). It is not explicitly stated that two events are assumed not to occur simultaneously, however, it is natural to make the assumption that no two events are allowed to occur simultaneously in ODE as distribution nor parallelism is not mentioned at all.

In the SAMOS platform [GD93] no explicit assumption is made concerning the possibility of simultaneous events. As SAMOS is not directed

towards parallel or distributed systems it is safe to assume that there are no simultaneous events.

In the REACH prototype [Deu94, BZBW95] no explicit assumption is expressed about simultaneous events, however, they claim that their event management could be distributed. Their event management is built on the same principles as Snoop [Deu94] and, to the author's knowledge, they do not allow simultaneous events.

Definition 8 Simultaneous events *are events that occurs at the same time according to the granularity of the chosen time scale, i.e., they receive the same time-stamp.*

Total Order of Events: It is possible to allow clients to make requests of a common server simultaneously and still order them sequentially by differentiating them in some other way. Examples of properties that have been used for differentiating requests for ordering purposes, are relative importance of the request, the causal precedence between requests (e.g., a file cannot be read before it is opened) , or a total ordering over the clients making the requests. Examples of these can be found in literature about operating systems [SG94]. For example, if two timestamped requests are generated with the same time-stamp simultaneously at different nodes in a distributed system they can be differentiated if all nodes are totally ordered (at that instant).

The same technique is applicable to the problem of total order of events because requests are ordered for correctness and fairness criterias and events must be correctly ordered. There will be a delay between the detection of two simultaneous events occurring at two separate nodes, which will be discussed later.

Definition 9 *A* **total order of events** *is assumed to exist, i.e., simultaneous events are differentiated according to some criteria.*

3.1.2 Primitive Events

Primitive events are in the SAMOS prototype [GD93] distinguished from *composite events*[1] in that primitive events are elementary occurrences, whereas composed events are composed of primitive and composed events using some *event algebra* (see section 3.1.3.

The specific primitive events for a given system differ, but as stated in REACH [BZBW95] the set of primitive events in an active object-oriented DBMS must contain: *arbitrary method invocation events* — which occur whenever a method is invoked; *temporal events* — which occur when the specified time is reached; and *control-flow events* — which occur whenever there is an important change of the control-flow in the database such as start of transaction, commit transaction, end of transaction etc.

In the SAMOS platform [GD92, GD93] and the Snoop prototype [CM93] the primitive event set is extensible because both projects follow the base extensible type schemas found in programming languages.

Definition 10 *A* **primitive event** *is a predefined elementary event that is generated by the system, by the environment, or by the application.*

Supported Primitive Events: There are variations between the considered active database prototypes in terms of which primitive event types are generated from the system. Examples only of primitive events are presented in ODE [GJ92a] which are subsumed by the other projects, hence they will not be separately discussed in this survey. There is general agreement of what primitive event types should be supported.

In an object-oriented DBMS most primitive event types can be modeled as *method invocation events* because most operations are imple-

[1] Also known as complex events.

mented as methods, however, they will be separated in this study because they are conceptually differentiated and cannot be modeled as method invocation events in a relational DBMS. No distinction will be made between method, procedure or function calls as the major difference is only in automatic instrumentation (see section 2.4.3).

In all prototypes, except ODE where other constructs are available, the begin-of and end-of event modifiers (see section 3.1.3) are available.

In this study the primitive event types are divided into: i) events generated by event modifiers applied to intervals (see section 3.1.3) — i.e., method invocations, *database operations* (operations performed on the database), and transactions; ii) *temporal events* — events generated by the clock [CM93]; iii) *explicit events* — events generated by the application (the user) [CM93]; and iv) *value events* — events generated when an attribute in an object is accessed or updated [GD92].

In the discussion below the taxonomy for the Snoop prototype is presented in figure 1 on page 14 [CM93], the taxonomy for the SAMOS prototype is presented in figure 2 on page 15 [GD93], the taxonomy for the REACH prototype is presented in figure 3 on page 15 [Deu94, Buc94], and, finally, the suggested taxonomy in this study is presented in figure 4 on page 16.

Method Invocation Events: In the Snoop prototype [CM93] the method invocation events are categorized as database events, however, our view is that if these can be generated by an application that is not accessing the database it is more accurate to not categorize them as database events — i.e., as is done in the SAMOS prototype [GD92, GD93] and the REACH prototype [Deu94, Buc94].

Database Events: In the Snoop platform [CM93], which is designed for both object-oriented and relational DBMSs, the database events consists of the events generated when the traditional database operations (i.e., insert, update, and delete) are executed and when transactions are started or completed.

In the SAMOS [GD92] and REACH [Buc94] platforms the database events are not present as they are considered to be method invocation events, however, this is conceptually bad according to our opinion because the taxonomy lacks clarity.

Our view is that database operations consists of insert, update and delete, however, as transactions can be a part of the operating system (e.g., Alpha Kernel [Nor87]) they are separated from the database events.

Transaction Events: In addition the *abort* event modifier were introduced to be able to trigger a rule if a transaction aborted in the SAMOS prototype [GD92]. Furthermore, the (begin to) *commit* event modifier were introduced in the REACH prototype to trigger rules before the transaction actually committed and to support coupling modes concerning detached transactions [Buc94] (see section 1.1.4). In this study the new event modifiers are considered strongly desirable.

Temporal Events: The *absolute temporal event* — i.e., an event occurring at a specified time — is considered to be a primitive event in all the discussed projects. However, the *relative temporal event* — i.e., an event occurring at a specified time relative to another event — is considered to be a composite event in REACH [Buc94] (see section 3.1.4).

The view in this study is that an absolute temporal event is primitive, whereas the relative temporal event is a composite event.

Explicit Events: Explicit events in the Snoop prototype [CM93] (a.k.a abstract events

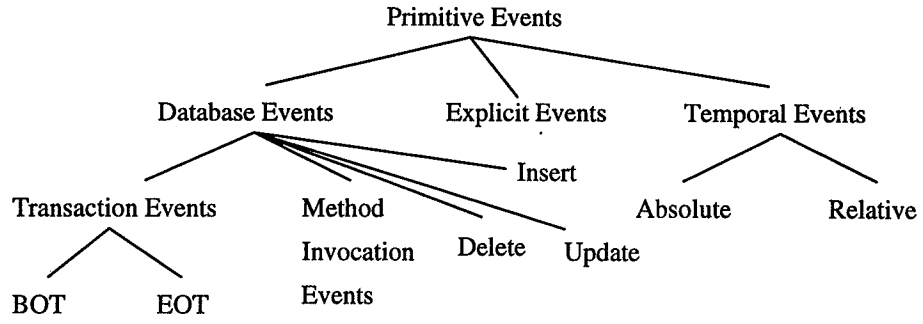

Figure 1: Primitive Event Taxonomy in Snoop

in the SAMOS prototype [GD92]) can be modeled as method invocation events, which is the reason for their absence in the REACH prototype [Buc94]. In this study it is considered more clear and understandable to express the explicit events in the taxonomy.

Value Events: Value events (cf ReadAttribute and WriteAttribute in the REACH prototype [Buc94]) were introduced in the SAMOS prototype [GD92] because it was possible to manipulate object attributes without calling a method.

If it is possible to manipulate object attributes without calling a method this event type is necessary for completeness. However, due to the problems of detecting this event type, our view is that object attribute access without calling a method should be restricted if possible, e.g., in the OBST DBMS [CRS+92] attributes can only be accessed through method calls.

Primitive Events for Distribution and Fault-Tolerance: None of the discussed active database prototypes explicitly take into account event types concerning distribution or fault-tolerance. For example, it is interesting to differentiate between data that is updated locally, and data that is propagated if rules are only to be triggered on local updates and not on propagated data. Another example is that if there are two active DBMSs with separate databases where one DBMS is a hot standby to the other DBMS, a DBMS crash can be described as an event type. When the primary DBMS crashed this can be detected by the hot standby.

The view taken in this study is that distribution and fault-tolerance event types are desirable, however, they are not present in the taxonomy as there is no work done on these event types.

Attributes of Primitive Events: The two natural attributes (see section 3.1.3) of a primitive event are event type ($type(E)$) and time of occurrence ($t_{occ}(E)$) [CM93], which are found in all discussed prototypes. In the SAMOS prototype the transaction identifier ($t_{id}(E)$) and the user identifier ($u_{id}(E)$) were introduced [GD92] in order to check if two or more events stems from the same transaction or user. This approach is adopted in the REACH prototype. The $t_{id}(E)$ and $u_{id}(E)$ are not explicitly mentioned in the early Snoop papers [CM93, CKAK93] as they were implicit, i.e., the event was detected within the transaction it was generated by the user who executed the transaction.

In this study the following views are held. The $t_{id}(E)$ and $u_{id}(E)$ can be generalized into the event's *scope* ($scope(E)$) — i.e., the currently active *dynamic scope* (e.g., a user, pro-

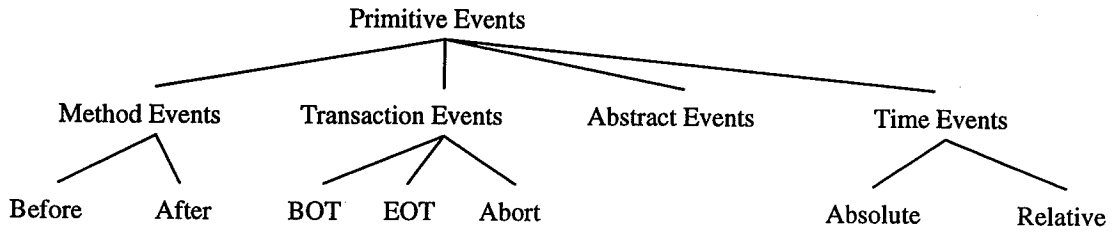

Figure 2: Primitive Event Taxonomy in SAMOS

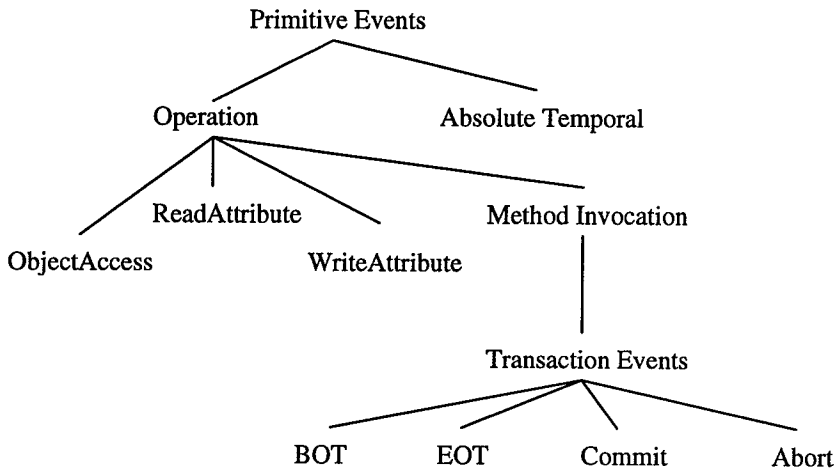

Figure 3: Primitive Event Taxonomy in REACH

cess, or a transaction) accessing a passive component (e.g., an object) — uniquely denoting where the event is generated. The *scope*(E) is architecture dependent, e.g., in a distributed system without distributed transactions where global event detection is needed it is necessary to let the scope include the node identity otherwise it is not possible for the global event detector to keep track of which node an event was generated on. Furthermore, the $u_{id}(E)$ can also be generalized into the *protection domain* of an event *dom*(E) — i.e., the domain specifying what resources are available to an active component (e.g., process, or transaction) under which conditions [SG94]. This gives the possibility to use all features found in the protection of resources.

Both the *scope*(E) and *dom*(E) is needed if the event monitor is not implicitly executing within the same scope and domain where the event was generated.

Definition 11 *The type(E) attribute denotes the event type of E.*

Definition 12 *The $t_{occ}(E)$ attribute denotes the time of occurrence of E.*

Definition 13 *The scope(E) attribute denotes uniquely where the event E stems from, i.e., what active component accessed what passive component.*

Definition 14 *The dom(E) attribute denotes what protection domain the event E was generated in.*

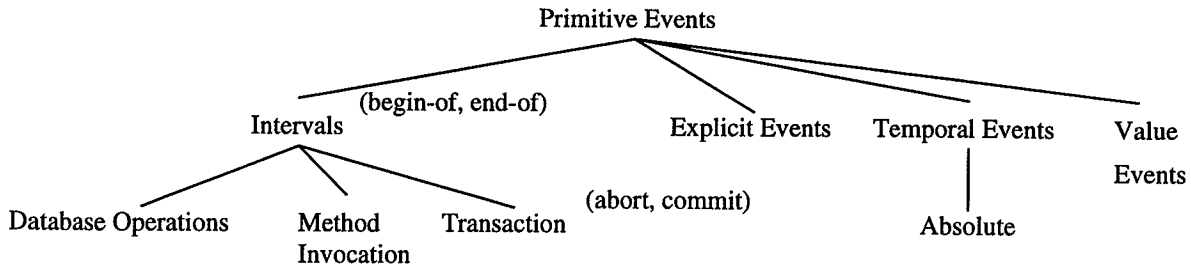

Figure 4: Resulting Primitive Event Taxonomy

3.1.3 Basic Definitions of Composite Events

Below general event history, event algebra and event expressions, initiator and terminator constituents, and event attributes and parameters are presented.

General Event History: In all the studied prototypes (ODE, Snoop, SAMOS, and REACH) an *event history* is basically a totally ordered set of event occurrences. The differences are mainly in the representation of event occurrences and the usage of the event history. Most of the work refers back to a general model based on the fact that the system has a complete history from a starting point which leads to highly inefficient algorithms [Deu94].

Obviously, it would not be possible to use the total order property in an event history if it would not be possible to differentiate and arbitrate between simultaneous event occurrences as mentioned in section 3.1.1.

Definition 15 *An* **event history** *is a totally ordered set of event occurrences; order is increasing on timestamps and preserves defined partial orders.*

Event Algebra and Event Expression: An *event algebra* is used to express *event expressions* (or *event patterns*). An event algebra consists of a set of event constructors, and a set of rules of how to form event expressions. The use of the term constructor stems from the fact that as events occur, composite event occurrences are detected by constructing them out of the event history using the constructors.

In Snoop [CM93] an event expression is informally an expression that defines an interval, i.e., a possibly non-instantaneous occurrence. The event constructor disjunction in Snoop [CM93, CKAK93] (and in SAMOS [GD92] and REACH [Buc94]) is exclusive and therefor expresses a point in time which, however, is a special case of an interval.

In ODE [GJ92a, JMS92, GJM93] an event expression is a mapping function from a event history to a subset of the event history using some predicate.

Definition 16 *An* **event expression** *(or* **event pattern***) is composed out of other composite or primitive events based on an event algebra using constructors.*

Henceforth, the term event expression is used.

Initiator and Terminator Constituents: In a composite event type the composition may be initiated by any member of a set of constituent types, the *initiator set*, where the actual initiating event is referred to as the *initiator* [CKAK93]. Similarly, the composition may be terminated by a member of a set of constituent types, the *terminator set*, where the actual ter-

minating event is referred to as the *terminator* [CKAK93].

These definitions are particularly of importance to event composition, furthermore, they make further definitions simpler.

Event Modifiers: In Snoop [CM93] *event modifiers* provide a mechanism for:

> "i) creating logical events at the conceptual level that correspond to points of interest in a closed interval and ii) mapping those points of interest to physical events in the system (i.e., suggest a feasible implementation). "

Currently two event modifiers are supported in Snoop: *begin-of* — the beginning of an event expression, e.g., beginning of a transaction; and *end-of* — the end of an event expression, e.g., end of method invocation, end of a sequence of three event types.

Care must be taken when applying event modifiers to arbitrary event expressions in a distributed system as it can suffer from independent failures. This means that applying the begin-of modifier to an arbitrary event expression can lead to the "detection" of an event that then fails to occur. For example, assume that a program, which can fail independently from the event detector, contain a sequence of method calls ($m_1..m_n$) and the event detector should detect the beginning of this sequence of method calls. If the program fails after m_1 and before m_n the event detector has detected an event that never occurred.

Definition 17 *An* **event modifier** *maps a point in a closed interval to an event.*

Event Type and Event Instance: In the SAMOS prototype [GD92, GD93] an event expression is used to describe the *event type* that will make a rule fire.

In this study an analogy to object-orientation is favored, i.e., an *event instance* is a representation of the actual occurrence of an event type.

Definition 18 *An* **event type** *is described by an event expression.*

Definition 19 *An* **event instance** *is an actual occurrence of an event of type event type and its parameters (see below).*

Event Attributes and Parameters: In the ODE prototype [JMS92, GJ92a, GJM93] and Snoop prototype [CM93, CKAK93], each primitive event is associated with a set of attributes called *event attributes.* They are referred to as *event parameters* in the REACH platform [Deu94] and more precisely as *actual event parameters* in SAMOS [GD92] [GD93].

In the SAMOS prototype [GD92] actual event parameters are distinguished from *formal event parameters* in the same way that actual and formal parameters of a procedure call are distinguished. Formal event parameters are used when specifying an event type and actual event parameters are bound to the formal event parameters at instantiation. The instantiation is performed as soon as the system is aware that the event has occurred; this may involve a delay after the actual occurrence, something which will be discussed later.

When an begin-of method event type occurs not all parameters may contain interesting information because some parameters may be output parameters whose values are only interesting after the method call has been completed.

In the ODE prototype [JMS92] event expressions are said to have parameters and the attributes of a composite event must be derived from the attributes of component event-subexpressions [GJM93]. Furthermore parameters are distinguished from attributes. Attributes only refer to the current primitive event, whereas parameters are attributes which

have been saved over one or more event occurrences. The explanation of the terms in ODE is not particularly clear.

In addition the Snoop, SAMOS and REACH prototypes all have a fixed set of event parameters that are associated with any event in the system and a set of optional event parameters that depend on the kind of the event.

An issue that is not mentioned is that the event attributes can be implicitly or explicitly carried with the event. An attribute that is implicit can be derived from the context in which the event instance is handled, whereas, an explicit event attribute cannot be derived. For example, if event monitoring only can be done within transaction boundaries then there is no need to make the transaction identifier attribute explicit because it can be derived from the context, i.e., the transaction.

Definition 20 *A fixed set of* **attributes** *describing the event is associated with every event type.*

Definition 21 *A* **formal event parameter** *is used as a part of the event pattern describing an event type.*

Definition 22 *An* **actual event parameter** *is an explicit or derived attribute of the event occurrence that is bound to the formal parameters of an event type during instantiation.*

3.1.4 Composite Events

In Snoop [CM93] a *composite event* is defined as an event obtained by the application of an event modifier — by default the end-of event modifier — to an event expression. An event expression is recursively defined as an event expression formed by using a set of primitive events, event operators and composite events constructed so far, i.e., its constituents.

In REACH a composite event is composed out of primitive events using event constructors [Deu94] and there is no explicit statement made about recursively composed events.

In ODE a composite event is specified as an event expression [GJ92a] or composite events can be combined out of basic events (cf primitive events) using logical operators and special event specification operators [GJM93].

Definition 23 *A* **composite event** *is specified using an event expression and occurs when the terminator event has been composed into a composite event instance.*

Supported Composite Events: All the considered active database prototypes support disjunction, conjunction, sequence and various forms *interval operators.* An interval operator is an operator defined over a closed interval of an event history with an initiator event type and a terminator event type defining the boundaries. There are distinctions concerning interval operators which will be discussed below.

Disjunction, Conjunction and Sequence: Our view is that all of these event constructors are needed.

The *disjunction* composition constructor stems from HiPAC [DBB+88] and is exclusive because simultaneous events does not exist on uni-processor systems, however, inclusive disjunction has been brought up lately because the possibility of simultaneous events in a distributed system [SHM95]. The computed parameters of an exclusive disjunction event are those of the particular constituent that occurred.

The *conjunction* event occurs whenever an arbitrary order of both constituent event types has occurred. In the Snoop prototype [CM93, CKAK93] there is a special variant of conjunction called *any* with which it is possible to detect an arbitrary sequence of a specified size and of specified event types. The parameters are com-

puted as the union of all parameters of the constituents.

The *sequence* constructor event occurs whenever an specified sequence of event types has occurred. The parameters computed using the union of the parameters of the constituents except for $t_{occ}(E)$, which is derived from the terminator constituent.

Interval Constructors: In Snoop [CM93] the following interval constructors are used: i) *aperiodic* — this constructor works as an enabling/disabling filter where every occurrence of an event type in a closed interval is instantiated and passed to the recipients (e.g., the rule manager in an active database) every time it occurs, e.g., if $aperiodic(E_1, E_2, E_3)$ is specified every occurrence of E_2 will be detected and passed to the recipients between the occurrence of E_1 event and an E_3 event; ii) *aperiodic** — the occurrence of the composite event type is instantiated when the terminator event type is instantiated and passed to the recipients, e.g., if $aperiodic*(E_1, E_2, E_3)$ is specified every occurrence of E_2 will be collected between the occurrence of an E_1 and E_3 event but the aperiod* event will not be detected until the occurrence of E_3; iii) *periodic* — every occurrence of a periodic event type in a closed interval is detected and passed to the recipients when it occurs, e.g., if $periodic(E_1, [t], E_2)$ is specified an event is generated with periodicity t between an occurrence of an E_1 and an E_2 event; iv) *periodic** — this constructor works like a sampler, i.e., the composite event type is instantiated when the terminator event type is instantiated. E.g., if $periodic*(E_1, X : [t], E_2)$ is specified the X will be sampled with a periodicity t between an occurrence of an E_1 and an E_2 event and the result will be instantiated and passed to the recipients when an event E_2 occurs; v) *not* — which is instantiated if and only if an event type does not occur within a closed interval, e.g., $not(E_1, E_2, E_3)$ is instantiated if an E_1 event occurs followed by an E_3 event without any E_2 events in between.

Unlike the other active database prototypes SAMOS [GD93] intervals are always closed by a temporal event type. SAMOS supports the interval constructors: i) the *closure* constructor — which stems from HiPAC prototype [DBB+88] and is the originator to aperiodic* operator and can be modeled like $aperiodic*(E_1, E_1, E_2)$; and ii) the *history* constructor — the composite event type is instantiated when a specified number of occurrences of an event type has occurred, e.g., the $TIMES(n, E)$ event is instantiated when the event E has occurred n times. This constructor may be viewed as a short form of a sequence. Furthermore, SAMOS also have the not constructor.

The REACH prototype supports the closure constructor, history constructor and the not constructor [Buc94, BZBW95].

As the Ode prototype supports a very expressive event specification language based on regular expressions all of the above constructors can be built out of their minimal set of basic constructors [GJ92a].

In this study aperiodic, aperiodic*, periodic, periodic*, not, and the history constructors are considered to be desirable due to the expressibility. However, these operators must be bounded in a real-time system otherwise it will be impossible to predict how large an event instance can be and, hence, it is not possible to calculate the time it takes to pass the event instance to the recipients.

Temporal Constructors: The periodic and periodic* operator in Snoop are also temporal operators. In REACH [Buc94] the relative temporal event is considered to be a composite event type, because only primitive events are possible to detect in an efficient manner and no real-time aspects can be applied on event composi-

tion in REACH. The relative temporal operator is viewed as a sequence with an additional complexity that a timer must be set when the initiator of the relative temporal event type occurs.

The REACH prototype [Buc94] introduces a new temporal event type *milestone* which is related to a relative temporal event and therefore considered to be a composite event type in REACH. The milestone event type is used to invoke contingency plans. For example, assume that a contingency action takes T_{cont} time and a transaction must be completed within T_{compl} time then the milestone event will occur at $T_{compl} - T_{cont}$ time so that the contingency transaction can be invoked. If the original transaction has not completed at T_{compl} it is aborted and the contingency transaction is committed instead.

Our view is that the relational temporal event is an composite event constructor. Furthermore, there is a need for the milestone event type in a real-time system with contingency plans.

3.2 Event Detection and Composition

Event detection is the mechanism that detects the occurrence of an event and disseminates the event instance to the recipients, e.g., the rule manager in an active DBMS. *Event composition* is the process of assembling composite events out of its constituents. A composite event type occurs when there is a fully composed event instance of that type and can, hence, be detected (see composite events section 3.1.4).

In SAMOS [GD92] user defined event types are *signaled* directly to the rule manager rather than passed via the event detector, which is claimed to be more efficient. The drawbacks are: i) user-defined event types cannot be part of a composite event types; and ii) all event types are not uniformly treated. Uniform treatment of event detection/composition makes it less complex.

In addition, relational monitors [Sno88] has been suggested as a general monitoring concept, however, these are inefficient (see section 2).

3.2.1 Time of Occurrence and Time of Detection

As mentioned in Snoop [CM93] there is a difference between the time of occurrence of an event and the time of detection of an event ($T_{delay} = T_{det} - T_{occ}$) that must kept sufficiently low. If the granularity of the timebase of the computer system is g_v then $T_{delay} < 2 * g_v$ in a real-time system with δ_t-precedence (see section 1.1.3). This difference quantifies the *timeliness*, i.e., how fast the system can respond to an event. In the Snoop prototype [CM93] the lower bound of the timeliness is considered important for contingency actions, i.e., how fast can a contingency action be invoked. However, in a real-time system the upper-bound is more interesting for predictability (and efficiency) reasons. Furthermore, the lower and upper bound on event detection should preferably be as close as possible to avoid low resource usage (and increase predictability).

In the Ode, SAMOS and REACH prototypes and in a later paper about Snoop [CKAK93] the difference between the time of occurrence and the time of detection of an event is considered to be sufficiently low. This assumption is good enough for non real-time uni-processor systems.

In an event-triggered real-time system the delay between the detection and the occurrence of an event cannot always be assumed to be sufficiently low as these systems are prone to event showers (see section 1.1.1). For example, if the burst of events in an event shower consists of B events and it takes t_{treat} and t_{treat} is larger than the interarrival time time to detect each of the event then the delay between the occurrence and the detection of the last event in the burst will be $B * t_{treat}$ [KV94].

3.2.2 Stable Events in Distributed Systems

In a distributed system there is always a delay between the occurrence of an event at one node and its detection at another node. In global event detection this issue lead to the question: "When can an event be consumed?" [CM93].

This problem is analogous to the *stable request* problem [Sch94, Sch95b, page 176-177]. Clients request replicas to perform services for them and requests are totally ordered according to a pre-defined scheme. The question here is "When can a request be serviced?".

When a replica has received requests with higher timestamps from all nodes than a pending request, then the pending request can be served given that all clients send their requests in strict order and a request is assumed to always arrive at the destination. In a similar manner event instances can be consumed when a node has received event instances with higher timestamps than a pending event instance from all other nodes. If the distributed system is supported by a real-time network with support for group protocols an event is stable after $2 * \pi + T_x$ time (where T_x is the delivery delay in the system), otherwise it may be necessary to send *null event instances* [SHM95], i.e., event instances that is only sent if no events has occurred during a period of time.

If the distributed system is not supported by a real-time network then a more computationally costly approach must be used, i.e., querying the other nodes of events, or introduce indeterminism into the system [SHM95].

3.2.3 Event Monitoring and Transaction Boundaries

There are two issues found in the literature concerning events and transaction boundaries: i) the event instance's *validity interval* — i.e., under which conditions an event instance is still valid for event composition — and ii) the *semantics* of composite events based on events from different transactions.

Validity Interval: An issue brought up by Buchmann et. al. [BZBW95] is that not all composite events that the event monitor is able to detect will ever be fully instantiated. For example, if the composite event type is $E_1 and E_2$ and an E_1 event occurs, leaving a half-composed event instance in the event monitor. Without clear semantics it is impossible to tell when the half-composed event should be removed, or when it should be completed, e.g., which E_2 event belongs to which half-composed event. In the REACH prototype [BZBW95] they implicitly express the validity interval by using transaction boundaries — i.e., any half-composed events are removed when a transaction is ended — or explicitly by specifying the validity time of a composite event type if the event type's constituents spans transaction boundaries.

Semantics: When the constituents of a composite event type can stem from two or more transactions, the semantics of the composite event type is not clear because the atomicity property of a transaction is broken. However, if the event monitor does not disseminate any information about any detected composite events spanning transactions until after all the transactions have committed the property of atomicity is not broken [BZBW95].

3.2.4 Event Parameter Contexts

In Snoop [CM93, CKAK93] four *event parameter contexts* — which control the event consumption policies and the parameter computation of the event composer — were defined: (most) *recent* context, *chronicle* context (also known as *chronological* context [Buc94]), *continuous* context and *cumulative* context. These are described below followed by a discussion.

Most Recent Context: The most recent context is considered to be usable in an environment where a high rate of sensor readings are entering the system and it does not matter if a few readings are missed.

During event composition constituents of partially instantiated composite events are overwritten whenever a more recent constituent is detected. All event instances that are possible constituents in a composite event type that are not possible initiator constituents are flushed from the event graph after a composite event instance has been composed.

Chronicle Context: The chronicle context is used when there is a correspondence between events which needs to be maintained, e.g., workflow management and testing/debugging.

The initiator and terminator pair is unique and whenever a composite event is detected the oldest initiator and terminator constituent pair is used to compute the parameters. All constituents in a composed event instance are then flushed from the event graph.

Continuous Context: In this context, each initiator of a composite event type starts the detection of that event type. A terminator event may complete one or more occurrences of the same event type. This context is especially useful for tracking trends of interest on a sliding time point governed by the initiator event. In this context an initiator will be used at least once for composition.

The major problem of this context is that it produces combinations of events of which some or all are of interest. It therefore adds more overhead to the system and requires more storage. Furthermore, in a real-time system the worst-case execution time of the event composition algorithm might be too pessimistic. For example, if the event monitor is part of the transaction and a deadline scheduler is used the resource usage may decrease due to the event monitor.

Cumulative Context: All events instances that can possibly belong to a composite are collected from the time the initiator occurs until the terminator occurs. When a terminator is found, all events making up the composite event instance are flushed from the event graph.

Evaluation of the Event Parameter Contexts: In his paper [Buc94] Buchmann states:

> "As a minimum, a system must support recent and chronological consumption policies"

The reason being that most systems can be designed without the continuous and the cumulative contexts. Furthermore, in a real-time system these contexts introduce too much unpredictability and overhead in the system.

Critique: The chronicle event parameter context has an underlying assumption of how events occurs in the system that is not applicable in all cases, e.g., if the sequence of opening and closing a file should be detected and file "A" is opened followed by an opening of file "B" followed by a closing of file "B" then a composite event instance consisting of the sequence open file "A" and close file "B" would be detected. More generally: i) event instances may be incorrectly composed out of unrelated event instances; and ii) a composite event instance may be lost because an occurrence of an event incorrectly replaces parts of half-composed event instances.

Solution: Current research is investigating the question of context sensitive conditional (filtering) event composition, i.e., it should be possible to state under which conditions an event composition should take place [Sch95b]. For example, if it would be possible to state that the

event instances should refer to the same file in the previous example then the problem can be solved.

3.3 Expressiveness vs Efficiency and Predictability

3.3.1 A Comparison of Internal Representations

In a comparison of composite event detection methods made by Deutsch [Deu94], where space requirements and the complexity of the algorithms are discussed, the results in table 1 on page 24 are noteworthy.

The table highlights: i) what type of *grammar* (arbitrary or operator) the event specification language is based upon; ii) the *space requirement* (e.g. number of nodes) of the internal representation for composite event detection; iii) the *transition cost* in the internal representation when a primitive event is instantiated; iv) whether the transition cost depends on the number of composite events that is specified, and v) if the internal representation of the event specification is suited for parallel or distributed composite event detection.

The comparison shows that, in terms of efficiency of the composite event detection algorithms, the event graphs are far better than finite state automata and Petri-nets. The transition cost of an event graph is constant and it requires considerably less space than a finite state automaton. Furthermore it is suited for parallel detection and composition.

The price that has to be paid for efficient composite event detection in a system, is less expressibility in the event specification language. However, in the Snoop [CM93, CKAK93] and REACH [Deu94, BZBW95] prototypes it is strongly indicated that it is possible to cover most situations that need to be detected with a simple yet powerful event specification language based on an operator grammar.

The ODE prototype [JMS92, GJ92a, GJM93] has a very expressive event specification language where conditions, which can access the whole database, are added as filters. These filters may make the event detection very inefficient and the use of (extended) finite state automata may result in a state explosion in the internal representation of the event specification.

Even though SAMOS, where Petri-nets are used as the internal representation [GD92, GD93, GGD94], has the possibility to use any grammar, an event operator based grammar is used.

3.3.2 Event Criticality

Event criticality denotes how critical an event is in an environment, which affects the event consumption policies, e.g., event types triggering associated critical tasks (e.g., rules) are assigned high criticality in order to have a higher probability of guaranteeing their deadlines — especially during event showers (see section 1.1.1). For the purpose of this study only *high* and *low* criticality will be considered because it is trivial to extend the examples.

Assigning Criticality to an Event Type: One way if assigning criticality to an event types is to map the criticality of a task's deadline to the task's associated event type(s). However, *criticality inheritance* and *parameter dependence* (explained below) threatens the concept of event criticality.

Criticality Inheritance: A constituent event type must be assigned the maximum criticality derived from its associated tasks and the composite event type(s) it is part of, otherwise detection of a critical composite event type may be delayed. This may, in the worst case, imply that all primitive event types are assigned high criticality.

Property	Internal Representation		
	Finite State Automata	Petri Nets	Event Graphs
event expression grammars:	arbitrary	arbitrary	operator only
number of internal nodes for n primitive events	$<= 2^n$	O(n), $> 3n$	O(n), $> n$
transition cost	1	$O(n^2)$	O(1)
transition cost depends on number of composite events	No	Yes!	No
suited for parallel detection	No	No	Yes

Table 1: Qualitative Comparison of internal representation of composite events

Parameter Dependence: The criticality of an event type may also depend on the parameters, e.g., a temperature reading event ($ETemp(T)$)of a monitored patient is not critical unless the temperature is above 43 degrees Celsius. This may also cause all primitive event types to be assigned high criticality.

A Solution: By using *extended event types* — which are the *basic event types* (see section 3.1.3) expressing criticality — that are either generated instead of the basic event type or transformed from the basic event type before the event detection not all primitive event types are assigned high criticality [BH95]. For example, instead of having one event type expressing the temp reading event there are two extended event type $ETemp_{High}(T)$, which only occurs when the temperature is above 43 degrees Celsius, and $ETemp_{Low}(T)$, which always occurs. As a consequence more events are generated and it remains to be seen if this technique can be used.

Implementation: The event criticality of an event instance must be determined before or when it is detected. This can be done two ways: i) by the event monitor that checks the criticality when the event instance is received — as this requires lookup tables event criticality may be dynamically changed, however, the overhead cost may be too large; and ii) the event instance is attributed with a criticality by the (software) sensor — a method that is more efficient than the first case, however, it is difficult to perform dynamic changes of the criticality.

In a real-time system the second approach is more appropriate as there is no need to handle unanticipated dynamic changes. A real-time system often operates in various *run modes* — a run mode is subset of all tasks that are activated during a certain situation, e.g., a flight control computer is in different modes during cruising, taking off, landing, and taxing — where the dynamic changes are between these prespecified run modes. These dynamic changes are possible to handle in the second approach as the amount of run modes normally are few, changes does not take place often, and a (software) sensor only needs to lookup the current run mode.

4 Debugging

Debugging normally involves step by step execution of sequential code, setting breakpoints, and tracing execution of code and changes of variables. This is straight-forward in a traditional non real-time sequential system, but not in a parallel, distributed, or real-time system [Gai85, Pla84].

Research in debugging can be categorized

as: i) control and observation — i.e., external control of the execution and monitoring of the execution; ii) visualization [Mil92]; and iii) techniques and tools for assisting debugging — e.g., *slicing* (a technique to indicate in what parts of the system where the fault may be) [Agr91, KSF92], or *algorithmic debugging* (fault diagnosis using additional information about the system) [Sha82].

After a general introduction the issues of control and observation will be discussed. The issues of visualization, as well as techniques and tools will not be covered, as they are considered not to affect the control and observation policies or mechanisms of the debugger.

4.1 Testing vs Debugging Activities

Testing and debugging are often confused [Bei90]. A common distinction is that the purpose of testing is to show the existence of faults, whereas the purpose of debugging is to localize the fault [Bei90, Som92].

4.2 Aspects of traditional debugging

Debugging normally follows the cycle of introducing a fault hypothesis, testing the fault hypothesis and localizing the fault (execute code) and edit code (hopefully correcting the error) [Agr91].

Debugging can be divided into: i) *logical debugging* — the purpose is to localize logical faults; ii) *performance debugging* — the purpose is to localize performance bottlenecks; and iii) *timeliness debugging* — the purpose is to localize timeliness faults [TKM89].

4.2.1 Functionality of Logical Debuggers

The functionality of traditional logical debugging is, typically, as follows: i) setting conditional or unconditional break points in the code — when a breakpoint is encountered the application is stopped and control of the system is passed to the debugger; ii) executing code step by step — this can be viewed as a special case of setting break points; iii) conditionally or unconditionally tracing execution of code or changes to variables; iv) inspecting variables and changing their values; and v) changing the flow of execution in the code — e.g., making an arbitrary jump to any point in the application code.

This functionality is independent of the *level of debugging*, e.g., machine code, assembly language code, or source language code such as ADA, C++ and LISP. The functionality is not related to whether or not the code is compiled or interpreted. It is simpler to build a debugger for a interpretive language and normally these debuggers are more flexible and have more expressive power than their counterparts for compiled programs [Agr91].

4.2.2 Functionality of Performance Debuggers

The goal of performance debugging is to localize bottlenecks, which requires a simpler functionality than in logical debugging. The major problem in performance debugging is visualization which is not covered in this survey. Performance debugging can be based on event traces, however, other methods are preferred as these traces may become immense in size.

4.2.3 Functionality of Timeliness Debuggers

The author knows of no actual debuggers for timeliness debugging, however, the ARTS operating system [TM89] includes the concept of *time fence* — a time assigned to an operation and if the operation does not finish within the time fence an exception will be raised — which can be used for timeliness debugging.

4.3 Control and Observation for Logical Debugging Purposes

In order to debug a system it must be instrumented (see section 2). Event monitoring (see section 3) can be used for debugging purposes, however in performance debugging it is not necessary to use event traces.

4.3.1 On-line vs Off-line Debugging

Debugging can either be done *on-line* — the debugger observes and controls the system while it is executing — or *off-line* — the debugging is performed *post-mortem*, i.e., after the system has been executed, on a event trace collected of the system

The advantage of on-line debugging is interactive control and ease of use, but the disadvantage is that it may cause a probe-effect (see section 2). This is serious even for non real-time parallel and distributed systems due to the unpredictable behavior, e.g., it may be impossible to reach the system state where an error occurs.

Off-line debugging is a way of avoiding or minimizing the probe-effect by debugging the system post-mortem, however, it is more difficult to use and in order to inspect the system in a certain state, *replaying* execution (see section 4.3.2) — i.e., re-executing the events in a trace in order to reach a certain state — is necessary.

4.3.2 Replaying Execution

Replaying is interesting in logical debugging (and possibly timeliness debugging). Replaying of an event trace for debugging purposes is equivalent to the problem of *regression testing*, i.e., the test suites should be re-executed after the system has been corrected in order to check that new faults were not introduced. The purpose of replaying for debugging differs from regression testing in that only one certain state needs to be reached.

The three major problems in replaying an execution are: i) the time to replay an event trace to a certain state may be unacceptably long (this is not a major issue in testing) [NSX94]; ii) in a real-time system the exact timings must be used during replay (not only the order) (cf regression testing [Sch94, Sch95a]); iii) additional probe-effect should not be added during replay [Sch94, Sch95a]; and iv) an environment simulator must be used in a real-time system [Sch94, Sch95a].

Furthermore, according to [Sch95a] the exact order and timing of *significant events* (i.e., synchronization of processes, access to the system's notion of time, and asynchronous interrupt) must be recorded to be able to correctly replay an execution. It is our contention that the primitive event taxonomy (see figure 4 covers most significant events in an active database.

Tracing for Fast Replaying: In the work of Netzer et. al. [NSX94] an approach mixing message logging with independent checkpointing of process states is presented which reduces the amount of logged messages by 90 percent compared to logging all messages. However, their tests only covered CPU and message intensive applications where the average reduction of message logging was measured.

5 Summary

5.1 Event Monitoring

5.1.1 Traditional Steps of Monitoring

The traditional (and general) steps in monitoring were introduced (section 2). These steps are typically followed when any kind of system is monitored (cf Instrumentation of Active Databases, section 2.4.3).

5.1.2 Monitor Implementations

There are software, hardware, and hybrid monitors where hybrid monitors combine the flexibility of a software monitor with the minimal intrusion caused by a hardware monitor (see section 2.1). Hence, it is possible to make the instrumented tasks more predictable as the event generation and the data collection part of the monitoring may be designed so that only a constant and small overhead is added to the task (see section 2.2).

Furthermore, a hybrid monitor can be used for monitoring by software engineering tools without causing any additional intrusion on the event generation and observation parts of the system. However, time to perform event detection increase because the time to disseminate the event instances to the recipients increases. This extra overhead due to multiple recipients must be considered during the design of the system. For example, if the system is supported by a real-time network with group protocols the time to disseminate the event instance to multiple recipients will be constant (see distribution section 1.1.3).

5.1.3 Event Specification

Concepts used in the Snoop, REACH, SAMOS and ODE prototypes were discussed in terms of fault-tolerance and distribution and the following definitions were adopted in this study (see section 3.1). The basic definition is that an event is atomic and instantaneous in contrast to an interval, which can stretch over a period of time. A logical event is the representation of an event on the conceptual level in contrast to a physical event (to which a logical event is mapped using an event modifier), which is the actual event that occurs when a logical event is said to occur. A filtered event is an event that occurs if and only if an associated condition (filter) holds true (this subsumes the term "logical event" as used in the ODE prototype). Simultaneous events are events that occurs at the same time according the granularity of the chosen time scale, however, it is possible to avoid the problems caused by detecting simultaneous events by differentiating according to a pre-defined order and, thus, preserve the total order.

An event modifier maps a point in a closed interval, e.g., transaction, to an event. An event type is described by an event expression, which is composed out of other composite or primitive events based on an event algebra using constructors. In this study an analogy to object-orientation is made, i.e., the actual occurrence of an event of type event type is an event instance. Event attributes are separated from parameters, which are used during composite event detection.

Furthermore, definitions for definite event, causal precedence, general event history, event algebra, initiator and terminator constituents, event attribute, event parameter were discussed.

Primitive Events: A primitive event is a predefined elementary event that is generated by the system, by the environment, or by the application. The suggested resulting taxonomy of primitive events (see figure 4 on page 16) in this study divides primitive events into: i) event generated by applying event modifiers to intervals, i.e., method invocations, database operations, and transactions; ii) temporal events only consisting of the absolute temporal event type; iii) explicit events; and iv) value events. The major reason for this taxonomy is clarity and, hence, even if most of these primitive events can be modeled as method invocation events this is not shown in the taxonomy. The more general concept of an interval is used to categorize events generated in closed intervals. Furthermore, transaction events are separated out from database events as there may be transaction systems outside the DBMS, i.e., a process executing

within a transaction may not access the database.

Primitive events for distribution and fault-tolerance may be needed, however, as there is no work done on these event types in active databases no such event types are included in the taxonomy.

The event modifiers begin-of, end-of, aborted, and (prepare to) commit are all considered necessary (see section 3.1.3). The first two are needed in order to map an interval into events. The abort modifier is needed in case it must be possible to trigger recovery rules if an transaction fails. The commit event modifier is needed to be able to trigger rules before the transaction actually commits, which may be used for two phase commit and various coupling modes.

The event attributes $type(E)$, $t_{occ}(E)$ (time of occurrence) are the two natural attributes of an event. Furthermore, in this study we present a generalization of $t_{id}(E)$ and $u_{id}(E)$ called $scope(E)$, which uniquely denotes which active component (e.g., process, transaction) is accessing which passive (component). The actual content of $scope(E)$ attribute is architecture dependent and is only necessary when the event monitor executes outside the scope where the event was generated. In addition, the protection domain $dom(E)$ attribute may have to be carried explicitly if it must be possible to protect resources during event monitoring. (See section 3.1.2).

Composite Events: A composite event is specified using an event expression and occurs when the terminator event has been composed into a composite event instance. (See section 3.1.4).

The disjunction, conjunction and sequence constructors are adopted in this study, as well as the interval constructors aperiodic, aperiodic*, periodic, periodic*, and not.

The history constructor can be viewed as a short form of sequence constructor and, thus, it is not strictly necessary. The closure operator can be modeled using the aperiodic* operator. Furthermore, the any constructor found in Snoop can be viewed as a short form of conjunction and, hence, it is not strictly necessary.

In addition the temporal constructors consists of relative temporal constructor and the milestone constructor (i.e., events generated in order to start contingency actions or to monitor the progress of a transaction).

5.1.4 Event Detection and Composition

Event detection is the mechanism that detects that an event has occurred, whereas event composition is the mechanism that composes composite event instances out of its constituents. A composite event occurs when it is fully composed and, hence, it can be detected.

In order to be able to perform global event detection and composition in a distributed system it must be possible to determine when an event instance is stable and can be consumed. If the distributed system is supported by a real-time network then an event instance is stable after $2 * \pi + T_x$ time after the occurrence of the event (where T_x is the delivery delay in the system). In other cases more computationally costly or insecure approaches be used which results in that either the event detection and composition will take a long time and unpredictability is introduced (see section 3.2.2).

Composite event detection using event graphs is a efficient and predictable mechanism. Furthermore, the event graphs requires less space than petri-nets and finite state automatas. In addition, event graphs are suited for parallel detection and composition (see section 3.3).

Validity intervals are used to determine when half-composed event instances are valid. If the event type does not span transactions boundaries then the validity interval is simply the lifetime of the transaction, otherwise the validity

interval must be specified explicitly (see section 3.2.3).

Event parameter contexts are used to specify the event consumption and event parameter computation policy, however, these are not enough as they only work under certain assumptions. As a result composite event instances may be lost or incorrectly composed out of constituents (see section 3.2.4).

5.1.5 Expressiveness vs Efficiency and Predictability of monitoring

The advantages of expressing what should be monitored using a declarative language (cf TQuel section 2, event specification languages section 3) are: i) the steps required to monitor a system will be reduced; ii) it is more understandable and, hence, maintainable — i.e., easier to change. However, the complexity of a general approach (cf declarative relational monitoring section 2, expressiveness vs efficiency and predictability section 3.3) makes it too unpredictable and inefficient. A more restrictive, but still powerful approach has been used in the event monitoring of the Snoop and REACH prototypes.

In order to make an event specification predictable the interval operators in the Snoop, REACH, SAMOS and ODE prototypes must be bounded, i.e., there must be limit to the amount of constituents that a composite event type "interval operator" may consist of (see interval operators section 3.1.4).

In order to assure predictable and efficient event detection — especially during event shower — for critical tasks the usage of event criticality has been suggested, i.e., the event monitor prioritizes critical events. This is, however, not without problems leading to that all event types in the system may be considered critical, e.g., if a composite event type is critical its constituents must inherit the criticality. By using extended event types expressing the criticality, whose instances are generated when the associated basic event type occurs under the correct conditions, it is possible to avoid these problems. At least an event instance of the lowest criticality will always be generated when the basic event type occurs. A problem with this solution is that more event instances are generated in the system (see section 3.3.2).

The time it takes to perform event composition and event detection must possible to derive from the event specification and be bounded. The interval operators (see section 3.1.4) must be bounded (see section 3.2.1).

5.2 Debugging

Event monitoring can be used for software engineering in general. Debugging is divided into logical debugging, performance debugging and timeliness debugging depending on the purpose. This survey concentrated on logical debugging, but the problems of using event traces are equivalent for all three kinds of debugging.

In a distributed real-time system the difference between debugging and testing is mainly in the purpose as off-line debugging is necessary to avoid the probe-effect, i.e., the purpose of testing is to show the existence of faults, whereas the purpose of debugging is to localize them. Hence, the mechanisms differ, however, both testing and debugging is started by collecting an event trace, which is the only general way of avoiding the probe-effect. After the event trace is collected it is analyzed in testing, whereas, in debugging replay of the event trace is necessary. The replaying of an event trace suffers the same problems as regression testing, i.e., replaying with the exact timings and order, and no additional probe-effect may be introduced during replay.

It is our contention that the resulting primitive event taxonomy covers most of the significant events that must be recorded to be able to

correctly replay an execution.

5.3 Future Work

The current work issues are: i) how to make event detection/composition predictable and efficient; ii) reviewing various testing techniques in order to investigate if the primitive and composite event types are sufficient; iii) investigate the interaction between the scheduler and an event monitor executing together on a CPU separated from the application; iv) reviewing the necessity of detecting simultaneous events; and v) evaluating conditional (filtering) event detection.

Acknowledgments

The author would like to thank Prof. Sten Andler for numerous discussions and extensive comments, Prof. Sharma Chakravarthy for discussions, valueable comments and help on event monitoring, Brian Lings for his valuable comments, and Mikael Berndtsson for his valuable help.

Appendix: OSE Delta

OSE Delta is a proprietary operating system developed by ENEA Data AB[OSG95, ORR]. A short description of the common properties concerning structure and portability, processes, communication, scheduling and modularity are described followed by the more interesting properties such as distribution [OSG95, ORR] and the Distributed Debug Server (DDS) [ODG95, ODR95].

Structure and Portability

The general strategy of the design OSE Delta is portability and efficiency, i.e., only a small part of the operating system is dependent on the hardware architecture while it handles interrupts in an efficient manner. This strategy can be found in other state-of-the art operating systems such as Chorus [RAA+90] and QNX [Hil92].

The interfaces between the operating system and hardware driver routines are open in order to let users and vendors supply their own drivers. Currently two major driver routines have a public an interface: i) the file system, and ii) the *link handler* — the link handler manages the communication links between the distributed nodes.

OSE Delta can execute on hard and soft targets. On a soft target preemption can only occur when operating system calls are executed and true interrupts cannot be handled, whereas on a hard target true preemption and interrupts can be handled (the kernel is protected but have preemption points for increased concurrency [SG94]). No further distinction will be made between soft and hard targets in this survey.

Scheduling and Processes

Scheduling

All process types in OSE Delta are scheduled according to a preemptive priority based policy using FIFO ordering on the same priority except on the lowest priority which is handled using round-robin scheduling [Han94, OSG95, ORR].

Process Types

OSE Delta supports four kinds of process types: i) *interrupt process* — a process which is started when its associated hardware interrupt occurs; ii) *timer process* — a process that is started periodically; iii) *prioritized process* — the basic process type; and iv) *background process* — a process executing on the lowest priority.

Priority and background processes can be created dynamically at run-time (*dynamic processes*) or configured into the operating system (*static processes*), whereas interrupt and timer

processes must be configured statically into the system.

Parent-child hierarchy

There is no parent-child relationship between a creating process and a dynamically created process, i.e., a child can exist without its parent, the child does not inherit any resources from the parent, and there is no risk for implicit cascading termination [SG94]. However, there are no implicit communication channels such as UNIX pipes.

Process States

The processes in a system can be in the following states: running, ready, blocked, suspended, and *intercepted* — an intercepted process is a suspended process that is controlled by another process for system level debugging purposes.

Priority Inversion

It is not possible to set priorities from another process and, therefore, most solutions [BW89] to the priority inversion problem cannot be applied.

Interprocess Communication and Synchronization

OSE Delta supports: i) *asynchronous message passing*, i.e., only the receiving process is blocked, extended with the possibility to set a timeout on the receive primitive and *selective waiting* — i.e., being able constrain what message types a process can receive in an receive operation, and ii) semaphores — both general and *fast semaphores* (only the owning process can wait on a fast semaphore).

Modularity

It is possible to emulate a *virtual node* [BW89] — a virtual node consists of active components (processes) and passive components (procedures and shared variables), which can only be accessed through the virtual node's common external interface — using the *block* concept, in which processes can be grouped, and the simple protection mechanism in OSE Delta. However, it is not possible to create a sub block inside a block.

Distribution

The link handler maintains communication and *logical channels* — a logical channel hides the physical channel to a process and makes the handling of remote interprocess communication almost transparent — between processes on different nodes.

The link handler can either be i) statically configured — during compilation or booting; or ii) dynamically configured — during run-time. Static configuration is fully transparent to the user, however, the static configuration is not fault-tolerant, i.e., it is not possible to statically configure what should happen when a logical channel is broken. Dynamic configuration is not fully transparent because there is no global naming schema. Furthermore, in order to make a fault-tolerant system the process requesting monitoring of a logical channel must be designed to handle a specific message which is sent from the link handler when ever a monitored logical channel is broken.

Distribution and Order

When distributed OSE Delta kernels are used, total ordering is not supported because the messages (OSE signals) are not time stamped, which will lead to certain global properties being impossible to detect. E.g., it is impossible to check for global states and to make correct algorithms [SG94, Lam78] (see section 1.1.3).

Distributed Debug Server

The *Distributed Debug Server* (DDS) is a system level debugger, i.e., the DDS is only concerned about processes, blocks and systems. and contains no data about the source level language. The DDS must be connected to an OSE Delta kernel using the same physical communication links that OSE Delta kernels themselves are using, i.e., no extra hardware is needed. The DDS shares features with source language level debugger, which will be explained below.

Structure

On each OSE Delta kernel there is a special monitoring process, which has a tight coupling with the kernel, i.e., data about the system can be transferred to the monitoring process in an efficient way. If an OSE Delta system on a node must be debugged the DDS must be connected to the monitoring process on the OSE Delta kernel executing on that node. The DDS is a front-end to the monitoring process, where the DDS issues commands to the monitoring process, which in turn performs the control and observation of the system.

The DDS can only be connected to one monitoring process at a time, however, it is possible switch between such processes. If more than one kernel has to be debugged simultaneously, two or more DDS instances can be executed simultaneously, each connecting to a different monitoring process. However, this has limited use as total order is not supported (see section 1.1.3).

Features

The DDS can instruct the monitoring process to: i) trace events — traces the event to a buffer that can be displayed analyzed post-mortem via the DDS; ii) monitor events — displays the event occurrence immediately via the DDS; and iii) *catching events*, i.e., set a conditional or unconditional breakpoints on events (more than one thread of execution can be halted).

Events are specified using event types which currently are: i) send/receive message; ii) create/dispatch/kill process; iii) system warm/cold start; iv) system/user error.

Scope of monitoring and catching: It is possible to specify a scope consisting of blocks and/or processes in order to constrain the overhead caused by the monitoring and catching features, i.e., only the processes and blocks within the scope will be subjected to monitoring or catching.

Tracing: It is possible to enable/disable tracing of a specific event type (see section 2). Furthermore, it is possible to specify under what conditions tracing should be enabled.

Sequences: It is possible to place combinations of trace, monitor and catch commands in sequences. Debugger commands in one sequence are valid as long as the debugger is using that sequence. In order to handle different sequences it is possible to make another sequence of debugger commands the current sequence (called "goto" in DDS).

Conclusions concerning OSE Delta

In order to avoid the priority inversion problem there is a definite need for being able to set the priority of another process.

A better support for distribution is desirable, i.e., support for global ordering, a global timebase, and a global naming schema (see section 1.1.3).

If the interface to the monitoring process would be public it could be used for event monitoring as described in sections 3 and 2.

References

[Agr91] Hiralal Agrawal. *Towards Automatic Debugging of Computer Programs.* PhD Thesis, Purdue University, August 1991.

[And79] Sten Andler. Predicate path expressions. In *Conf. Rec. of the 6th Annual ACM Symp. on Principles of Prog. Lang.*, pp 126–136, January 1979.

[And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, 1991.

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[Bei90] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 2nd edition, 1990.

[BH83] Bernd Bruegge and Peter Hibbard. Generalized path expressions: A high level debugging mechanism. In *Software engineering symposium on high-level debugging*, pp 34–44. ACM SIGSOFT/SIGPLAN, 1983.

[BH95] Mikael Berndtsson and Jörgen Hansson. Issues in active real-time databases. In *Proc. Int'l Workshop on Active and Real-Time Database System (ARTDB-95).* Springer-Verlag, 1995. To be published.

[BR93] Özalp Babaoğlu and Michel Raynal. Specification and detection of behavioral patterns in distributed computations. Technical Report 93-11, Laboratory for Computer Science Research, University of Bologna, May 1993.

[Buc94] Alejandro P. Buchmann. Active object systems. In A. Dogac, M. T. Ozsu, A. Biliris, and T. Sellis, (eds), *Advances in Object-Oriented Database Systems*, pp 201–224. Springer-Verlag, 1994.

[BW89] Alan Burns and Andy Wellings. *Real-Time Systems and Their Programming Languages.* Addison-Wesley, 1989.

[BZBW95] A. P. Buchmann, J. Zimmermann, J. A. Blakeley, and D. L. Wells. Building an integrated active OODBMS: Requirements, architecture, and design decisions. Data Engineering, 1995.

[CGG94] Augustoa Ciuffoletti, Federica Gattai, and Roberta Golinelli. Clock synchronization in virtual rings. In *Proc. 6th EuroMicro Workshop on Real-Time Systems*, pp 72–77. EuroMicro, 1994.

[Chr89] F. Christian. Probabalistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[CKAK93] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Anatomy of a composite event detector. Technical Report UF-CIS-TR-93-039, Computer and Information Sciences Dept, University of Florida, 1993.

[CM93] S. Chakravarthy and Deepak Mishra. An event specification language (Snoop) for active databases and its detection. Technical Report UF-CIS-93-007, Computer and Information Sciences Dept, University of Florida, September 1993.

[CRS+92] Eduardo Casais, Michael Ranft, Bernhard Schiefer, Diet-

mar Theobald, and Walter Zimmer. OBST—an overview. Technical Report FZI.039.1, Forschungszentrum Informatik (FZI), Karlsruhe, Germany, 1992.

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, S. Chakravarthy, M. Hsu, R. Ladin, D. McCarty, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauharu. The HiPAC project: Combining active databases and timing constraints. *ACM Sigmod Record*, 17(1), March 1988.

[Deu94] Alin Deutsch. Method and composite event detection in the "REACH" active database system. Master's thesis, Technical University Darmstadt, July 1994.

[Gai85] Jason Gait. A debugger for concurrent programs. *Software-Practice And Experience*, 15(6), June 1985.

[Gal9x] Antony Galton. Instantaneous events. Technical report, Dept. of Computer Science, University of Exeter, 199x.

[GD92] S. Gatziu and K. R. Dittrich. SAMOS: An active object-oriented database system. *IEEE Data Engineering, Special issue on active databases*, 15(1-4):23–26, December 1992.

[GD93] Stella Gatziu and Klaus R. Dittrich. Events in an active object-oriented database system. Technical Report 93.11, Informatik der Universität Zürich, 1993.

[GGD94] Stella Gatziu, Andreas Geppert, and Klaus R. Dittrich. The SAMOS active DBMS prototype. Technical Report 94.16, Informatik der Universität Zürich, 1994.

[GJ92a] N. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proc. of the ACM SIGMOD Int'l Conf Management of Data*, pp 327–336, September 1992.

[GJ92b] N. H. Gehani and H. V. Jagadish. Active database facilities in ode. *IEEE Data Engineering, Special issue on active databases*, 15(1-4), December 1992.

[GJM93] N. H. Gehani, H. V. Jagadish, and Inderpal Singh Mumick. Temporal queries for active database support. In *Proc of Int'l Workshop of an Infranstructure for Temportal Databases*, Arlington, June 1993.

[GJS93] N. H. Gehani, H. V. Jagadish, and O. Schmueli. COMPOSE - A system for composite event specification and detection. In *Advanced Database Concepts and Research Issues*. Springer-Verlag, 1993.

[Gor91] Michael M. Gorlick. The flight recorder: An architectural aid for system monitoring. *SIGPLAN notices*, 26(12):175–183, 1991.

[Han94] Jörgen Hansson. Dynamic real-time scheduling for OSE Delta. Technical Report HS-IDA-TR-94-007, Department of Computer Science, Box 408, University of Skövde, September 1994.

[Hil92] Dan Hildebrand. An architectural overview of QNX. In *Proc. Usenix Workshop on Micro-Kernels & Other Kernel Architectures*, Seattle, WA, April 1992.

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Comm. ACM*, 17:549–557, October 1974.

[HW90] Dieter Haban and Dieter Wybranietz. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE Trans. on Software Engineering*, 16(2):197–211, February 1990.

[JMS92] H. V. Jagadish, I. S. Mumick, and O. Schmueli. Events with attributes in an active database. *AT&T Technical Memorandum 11356-921214-18TM*, 1992.

[JRR94] Farnam Jahanian, Ragunathan Rajkumar, and Sitaram C. V. Raju. Runtime monitoring of timing constraints in distributed real-time systems. *Real-Time Systems*, 7(3), November 1994.

[KDM88] A. M. Kotz, K. R. Dittrich, and J. A. Mülle. Supporting semantic rules by a generalized event/trigger mechanism. In *Proc. 18th Int'l Conf. on Very Large Data Bases.* Vancouver, 1988.

[KSF92] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzon. Interprocedural dynamic slicing. In *Proceedings of PLILP'92-Symposium on Programming Language Implementation and Logic Programming*, pp 370–384. Springer-Verlag, 1992.

[KV94] Hermann Kopetz and Paulo Veríssimo. *Real Time and Dependability Concepts*, Ch. 16, pp 411–446. Addison-Wesley, distributed systems 2 edition, 1994.

[Lam78] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communcations of the ACM*, 21(7):558–565, July 1978.

[Mil92] Barton P. Miller. What to draw? when to draw? an essay on parallel program visualization. Technical Report TR 1103, Computer Sciences Department, University of Wisconsin-Madison, June 1992.

[Nor87] J. Duane Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel.* Academic Press, 1987.

[NSX94] Robert H. B. Netzer, Sairam Subramanian, and Jian Xu. Critical path-based message logging for incremental replay of message passing programs. In *14th International Conference on Distributed Computing Systems*, pp 404–413. IEEE Computer Society Press, June 1994.

[ODG95] Distributed debug server R2.0 getting started & user's guide. Part of OSE Delta distribution, ENEA DATA AB, OSE Support Group, Box 232, S-183 23 Täby, Sweden, 1995.

[ODR95] Distributed debug server R2.0 reference manual. Part of OSE Delta distribution, ENEA DATA AB, OSE Support Group, Box 232, S-183 23 Täby, Sweden, 1995.

[ORR] Ose delta real time kernel 68k R1.3 reference manual. Part of OSE Delta distribution, ENEA DATA AB, OSE Support Group, Box 232, S-183 23 Täby, Sweden.

[OSG95] Ose delta soft kernel R1.0 getting started & user's guide. Part of OSE

Delta distribution, ENEA DATA AB, OSE Support Group, Box 232, S-183 23 Täby, Sweden, 1995.

[Par86] Derek Partridge. *ARTIFICIAL INTELLIGENCE applications in the future of software engineering.* Ellis Horwood, 1986.

[Pla84] Bernhard Plattner. Real-time execution monitoring. *IEEE Transactions on Software Engineering*, SE-10(6), November 1984.

[RAA+90] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS distributed operating system. Technical Report CS-TR-90-25, Chorus systémes, 1990.

[Sch94] Werner Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, September 1994.

[Sch95a] Werner Schütz. *The Testability of Distributed Real-Time Systems.* KLUWER ACADEMIC PUBLISHERS, 1995.

[Sch95b] Wieland Schwinger. Logical events in ECA-rules. MSc Thesis under preparation. University of Skövde, 1995.

[SG94] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts.* Addison-Wesley, 4 edition, 1994.

[Sha82] E. Y. Shapiro. Algorithmic program debugging. *MIT Press*, May 1982.

[SHH87] M. Stonebraker, E. Hanson, and C. H. Hong. The design of the POSTGRES rules system. In *Proc. 3rd IEEE*, pp 365–374, December 1987.

[SHM95] Scarlet Schwidersky, Andrew Herbert, and Ken Moody. Composite events for detecting behaviour patterns in distributed environments. In *Distributed Object Management 95*, July 1995.

[Sno88] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2), May 1988.

[Som92] Ian Sommerville. *Software Engineering.* Addison-Wesley, 4th edition, 1992.

[TKM89] Hideyuki Tokuda, Makoto Kotera, and Clifford W. Mercer. A real-time monitor for a distributed real-time operating system. *SIGPLAN notices*, 24(1), 1989.

[TM89] Hideyuki Tokuda and Clifford W. Mercer. ARTS: A distributed real-time kernel. *ACM SIGOPS Operating Systems Revies*, 23(3):29–53, 1989.

[Ver94a] Paulo Veríssimo. Ordering and timeliness requirements of dependable real-time programs. *Real-Time Systems*, 7(2):105–128, September 1994.

[Ver94b] Paulo Veríssimo. Real time communication. In Sape Mullender, (ed), *Distributed Systems*, Ch. 17, pp 447–490. Addison-Wesley, 2nd edition, 1994.

[You82] S. J. Young. *Real Time Languages: Design and Development.* Ellis Horwood, 1982.