Implementation Strategies for Time Constraint Monitoring

(HS-IDA-EA-99-108)

Sanny Gustavsson (a96sangu@ida.his.se)

Institutionen för datavetenskap
Högskolan i Skövde, Box 408
S-54128 Skövde, SWEDEN

Final year project in computer science, spring 1999.

Supervisor: Jonas Mellin

Implementation Strategies for Time Constraint Monitoring

Submitted by Sanny Gustavsson to Högskolan Skövde as a dissertation for the degree of B.Sc., in the Department of Computer Science.

[990618]

I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has previously been conferred on me.

Signed: _____

Implementation Strategies for Time Constraint Monitoring

Sanny Gustavsson (a96sangu@ida.his.se)

Abstract

An event monitor is a part of a real-time system that can be used to check if the system follows the specifications posed on its behavior. This dissertation covers an approach to event monitoring where such specifications (represented by time constraints) are represented by graphs.

Not much work has previously been done on designing and implementing constraint graph-based event monitors. In this work, we focus on presenting an extensible design for such an event monitor. We also evaluate different data structure types (linked lists, dynamic arrays, and static arrays) that can be used for representing the constraint graphs internally. This is done by creating an event monitor implementation, and conducting a number of benchmarks where the time used by the monitor is measured.

The result is presented in the form of a design specification and a summary of the benchmark results. Dynamic arrays are found to be the generally most efficient, but advantages and disadvantages of all the data structure types are discussed.

Keywords: Event Monitoring, Real-Time Systems, Time Constraints, Time Constraint Graphs

Contents

1 Introduction

All definitions of a real-time system have at least two things in common – the system must be able to not only deliver the correct results, but also deliver the results at the correct time. Also, real-time systems must be able to detect and react to what happens in the system environment.

These aspects of a real-time system, combined with the fact that real-time systems often operate in environments where uncontrolled or erroneous behavior of the system can have catastrophic consequences, mean that testing and run-time surveillance of the systems are of utmost importance. To facilitate testing and monitoring of real-time systems, tools such as event monitors can be used. An event monitor collects and processes information about the real-time system and its environment, and notifies the real-time system of any interesting results.

This dissertation discusses an event monitoring approach that uses graphs to represent the time constraints posed on the system. A general design for such a system is presented, and different configurations of the design are evaluated. The dissertation is disposed as follows.

Chapter 2 contains a background to real-time systems and event monitoring in general, and the graph-based approach in particular. Chapter 3 formulates the purpose and focus of this project, while chapter 4 describes the method used to reach the project goals. This chapter contains a detailed description of the design for the monitor. Chapter 5 discusses the obtained results, and chapter 6 contains concluding remarks about the project. The three first appendices (A-C) cover details about the event monitor implementation, and Appendix D contains brief discussion on how this work is connected to the theory of science.

2 Background

This chapter describes the preliminaries to this project, and also defines the concepts which are used in this dissertation. Section 2.1 gives a general description and definition of real-time systems, including relevant categorizations. Section 2.2 covers event monitoring, and defines the key concepts in that area. It also gives examples of areas where event monitoring can be useful. In section 2.3, a brief description of a language that can be used to specify constraints on a real-time system's behavior is given, and section 2.4 explains how run-time checking of these constraints can be facilitated by using directed weighted graphs to represent the constraints. Section 2.5 presents the objectives that should be attained by an event monitor. Finally, section 2.6 covers algorithm optimization and presents some optimization guidelines.

2.1 Real-time systems

There are several definitions of real-time systems, see for example Kopetz and Veríssimo (1994). The main commonality of all definitions is that a real-time system should not only produce correct results (functional correctness), but also produce these results at the correct time (timeliness). In many cases, real-time systems operate in environments where the inability to execute a task at the correct time could cause great losses, perhaps in the form of damaged equipment, or even human lives. For example, if a space shuttle fails to detach a segment at the correct time during the launch sequence, then the extra weight could mean that the shuttle is unable to exit the atmosphere. This would, in turn, probably result in the death of the shuttle crew and the destruction of billions of dollars worth of equipment.

The need for timeliness implies that all tasks in a real-time system must be predictable and sufficiently efficient. A task is *predictable* if there is a bound on the amount of resources that the task needs to complete. A task is *sufficiently efficient* if its resource requirements can be satisfied by the system, and if it is schedulable (i.e., can meet its deadline) given all other tasks in the system.

Real-time systems can be categorized into hard and soft systems (Burns and Wellings 1989), depending on how important it is that the system meets its deadlines, i.e., produces results in time. In a *hard* real-time system, deadlines must be met, or the system fails completely. In a *soft* real-time system, a result may be useful even though it was not delivered in time.

2.2 Event monitoring

An event monitor provides a way of observing what is actually happening in a real-time system. The monitor collects information about state changes in the system in the form of events. In the approach presented by Liu and Mok (1997a) this information is checked against a set of time constraints (see section 2.2.4) in order to decide whether the system conforms to its specification or not. Figure 2.1 gives a schematic view of a monitored system.

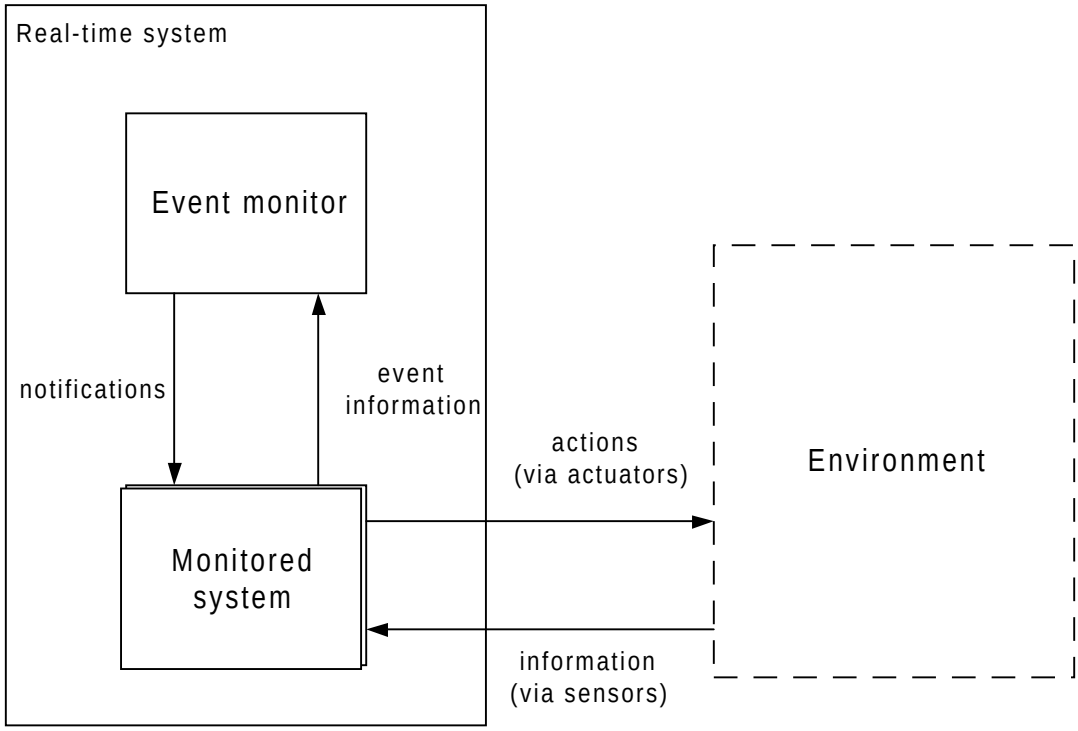

Figure 2.1: Overview of a monitored real-time system

The monitored system in this figure is the part of the system which interacts with the environment via sensors and actuators. The event monitor gains event information from the monitored system, and returns information (to the monitored system) in the form of notifications. For example, if a violation of a time constraint is detected by the monitor, a notification can be sent to the monitored system which then can take proper actions.

2.2.1 Primitive events

A *primitive event* can be defined as a change of the system state (Mok and Liu 1997b). According to the definition given by Chakravarthy and Mishra (1993) this state change is atomic i.e., either it happens completely or not at all and happens instantaneously, i.e., has no duration. Moreover, events are only mapped to state changes of interest, since the set of events for a given system would otherwise be arbitrarily large. Events can be defined to occur at, e.g., the start of a transaction, the sending of a message, or the termination of a process.

The difference between the terms event type and event instance should be clarified. An *event instance* is what is actually generated when a state change of interest occurs. All event instances that are generated by the same type of state change (e.g., when a message is sent) are defined to be of the same *event type*. In this dissertation, the term event is used when referring to 'primitive event instance', unless otherwise is stated.

Event types are generally *recurrent*, which means that several instances of an event type can occur during a computation (Mok and Liu, 1997a). An *event occurrence* is defined by Mok and Liu (1997a) as a point in time at which an instance of an event happens.

Each event type has an *event history*, which contains the instances of the event type (Chodrow, Jahanian, and Donner, 1991). Event histories are a necessary requirement for run-time monitoring of time constraints, since time constraints can refer to past event instances (further discussed in section 2.2.4). A bound on the length of the history for each event type can be extracted from the time constraint specification (Mok and Liu, 1997a).

2.2.2 Composite events

In the event specification language Snoop (Chakravarthy and Mishra, 1993), *composite events* are defined as events that are formed by applying a set of operators to primitive and composite events. Examples of such operators in Snoop are disjunction (denoted by "$\vee$") and sequence (denoted by ";"). The composite event (Abort_transaction $\vee$ End_transaction) occurs when either of the events Abort_transaction or End_transaction occurs. The composite event (Open_file ; Write_to_file ; Close_file) occurs when the event Close_file occurs, provided that the composite event (Open_file ; Write_to_file) has already occurred.

Composite events occur when an instance of a *terminator* event occurs (Chakravarthy et al., 1994). The terminator event is an event that denotes that a composite event has been completed. For example, the terminator of the sequence (Open_file ; Write_to_file ; Close_file) is the event Close_file. The terminator of (Abort_transaction $\vee$ End_transaction) can be either Abort_transaction or End_transaction. Conversely, the *initiator* event is the event that starts the composition of a composite event.

2.2.3 Event parameters

Each event type defines a set of *event parameters*, which are used to hold information related to each instance of the event type. The minimum set of parameters for an event instance is the occurrence time and the event type, since this information is needed to compile an event history for each event type (Chakravarthy and Mishra, 1993). For example, the parameter set for an event of type Car_passed in a traffic surveillance system could include the speed at which the car passed in addition to the time of occurrence and the event type. Each instance of an event type has its own set of values for the event parameters.

2.2.4 Time constraints

Time constraints can be used to specify time limits for the behavior of a real-time system. Time constraints can be in the form of task deadlines (e.g., the air-bag in a car must be inflated within 5μs when a collision is detected) or minimum delays between events (e.g., the WALK light at a zebra crossing should not be lit until at least two seconds have passed since the traffic light turned red). In our work, time constraints are specified using a language defined by Mok and Liu (1997a) which in turn is a refinement of real-time logic (see section 2.3).

2.2.5 Applications for an event monitor

There are two main applications for which an event monitor is useful. Those are *(i)* testing and debugging of real-time systems (Schütz, 1994) and *(ii)* supporting fault tolerance in a system (Mellin, 1998).

Testing and debugging

When testing and debugging a real-time system, timeliness must be considered. In addition, the correctness of an execution in a real-time system depends on what happens in the environment where the system operates during the execution. Thus, a way of observing a system's internal states is needed. An event monitor simplifies the observation of a system by detecting events and thus allowing event histories to be created during system execution.

If a bug is detected during a test execution, then it is often useful to be able to recreate this execution during subsequent tests to ensure that the bug has been removed. This kind of testing, called regression testing (Adrion, Branstad, and Cherniavsky, 1982), requires a log of the execution to be reproduced. For such a log, event histories can be useful.

Fault tolerance

Laprie et al. (1994) define a *fault* as a cause of an error, which is intended to be avoided or tolerated. Furthermore, they define *fault prevention* as the act of preventing a fault occurrence or introduction. This can be compared to *error detection and recovery*, which means that error recovery is done after the error has occurred and been detected.

An event monitor can be used to detect violations and satisfactions of time constraints. If a violation occurs, the monitor can notify the system of the error, and recovery measures can be taken. In some cases, the monitor can also predict future constraint violations, which could be useful for fault prevention.

2.3 Time constraint specification for event monitoring

The language used by Mok and Liu (1997a) to specify time constraints employs two main functions – the *event occurrence function* (denoted by "@") and the *index function* (denoted by "#").

Definition 1: *For an event e and a positive integer i, the event occurrence function is defined as*

$@(e,i) = t$

where t is the occurrence time of the i:th instance of event e.

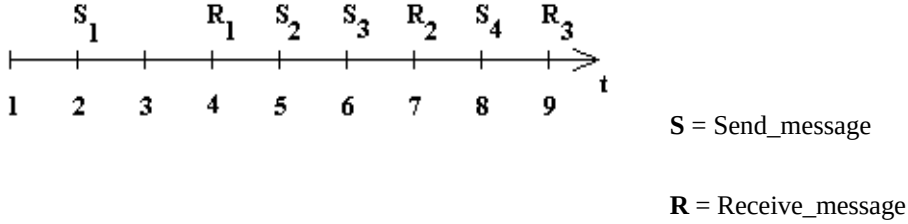

Figure 2.2: Event history example

For example, @(Send_message,3) returns the system time at the third occurrence of the event type Send_message. The most recent occurrence of the specified event is denoted by i=-1, the second most recent occurrence by i=-2 and so forth. An index value of 0 is undefined. Given the event history in figure 2.2, @(Send_message,3) returns 6, since that is the occurrence time of the third instance of the event type Send_message.

Definition 2: *For an event e and a time t, the index function is defined as*

$\#(e,t) = i$

where i is the index of the most recent occurrence of event e at time t. That is,

$$\#(e,t) = \begin{cases} 0 & \text{if } t < @(e,1) \\ i & \text{if } i \geq 1 \wedge @(e,i) \leq t \wedge @(e,i+1) > t \end{cases}$$

For example, #(Receive_message,5) returns the index of the last occurrence of the event type Receive_message at time 5. Given the event history in figure 2.2, #(Receive_message,5) returns 1, since the most recent occurrence of Receive_message at time 5 is the first occurrence, which happens at time 4.

The event occurrence function can be extended to represent event occurrences relative to a point in time. This extension is called the *relative event occurrence function* and is denoted by $@_r$.

Definition 3: *For an event e, a time t and an integer i, the relative event occurrence function is defined as*

$@_r(e,t,i) = @(e,\#(e,t) + i)$ *when* $\#(e,t) + i > 0$

For positive values of i, this function returns the time for future instances (relative to the specified time). For example, $@_r(e,5,1)$ returns the occurrence time of the next instance of event e at time 5. Given the execution depicted in figure 2.2, $@_r$(Send_message,4,2) returns 6, since #(Send_message,4) = 1 and @(Send_message, 1+2) = @(Send_message,3) = 6. The function call $@_r$(Send_message,@(Receive_ message,2),0) also returns 6, since the most recent event occurrence of the Send_message type at time 7 (the occurrence time of the second instance of Receive_message) happened at time 6.

Given these functions, basic time constraints can be written on the form $T_1 + D$ *op* T_2, where D is an integer constant, *op* is one of {>, ≥} and T_1, T_2 are event instance occurrence times represented by 0 or any of the functions @ and $@_r$. For example, the constraint @(Begin_transaction,i) + 10 ≥ @(Commit,i) expresses that the Commit event corresponding to a Begin_transaction event must occur within 10 time units.

Time constraints are composed by using disjunctions of conjunctions of basic time constraints (i.e., disjunctive normal form). For example, the following time constraint can be used to check that a transaction is either committed or aborted within 10 time units:

$(@(\text{Begin\_transaction},i) > 0 \wedge @(\text{Begin\_transaction},i) + 10 \geq @(\text{Commit},i)) \vee$

$(@(\text{Begin\_transaction},i) > 0 \wedge @(\text{Begin\_transaction},i) + 10 \geq @(\text{Abort},i))$

Constraint conjunctions can be optimized by removing *unnecessary constraints* (Mok and Liu, 1997b), which are constraints that can be removed from a constraint conjunction without affecting the semantics of the conjunction. That is, if a constraint conjunction that contains an unnecessary constraint is violated at a time *t*, then the same conjunction with the unnecessary constraint removed would still be violated at time *t* (the same can be said for satisfaction of constraints). Mok and Liu (1997b) describe algorithms for optimizing constraint conjunctions by removing unnecessary constraints.

2.3.1 Expressing composite events with time constraints

The specification language presented in the previous section can also be used to specify composite events, as shown by Liu, Mok, and Konana (1998). However, these specifications differ from those done in, for example, Snoop, since Snoop has a more restricive way of specifying relations between constituents of a composite events.

As an example, assume that the composite Snoop event SE = (Receive_message ; Send_message ; Receive_message) should be detected in the execution depicted in figure 2.2. At time 7, the event has occurred. However, it is not clear whether one or two instances of the event has occurred. Furthermore, if only one instance is detected, which of the Send_message instances (at times 5 and 6) should be seen as part of the composite event? In order to increase the expressiveness of Snoop, parameter contexts (Chakravarthy and Mishra, 1993) are used. The purpose of parameter contexts is to dictate when and how a composite event will be generated.

The parameter contexts defined in Snoop are *recent*, *chronicle*, *continuous*, and *cumulative* (Chakravarthy and Mishra, 1993):

- **Recent:** In the recent context, the most recent instance of the composite event's initiator event is used for composing the composite event. The composite event SE, for example, would be constructed as (R_1 ; S_2 ; R_2) at time 7, and as (R_2 ; S_4 ; R_3) at time 9. The event (R_1 ; S_2 ; R_3) is not detected, however, since the event composition that started with the occurrence of R_1 is flushed when the second instance of the initiator event (R_2) occurs. Note also that the event (R_1 ; S_3 ; R_2) cannot be detected with this context.
- **Chronicle:** If the chronicle context is used, a composite event is composed of the first instance of each event type. These instances are then ignored for the purpose of constructing future instances of the event. At time 7, SE would be constructed as (R_1 ; S_2 ; R_2) under the chronicle context, and the next instance of SE would be (R_2 ; S_4 ; R_3), not (R_1 ; S_2 ; R_3).
- **Continuous:** Under the continuous context, each instance of an initiator event is kept track of as a potential composite event. Given the execution depicted in figure 2.2, the composite event CE = ((Send_message $\vee$ Receive_message) ; Send_message) would be detected twice at time 5 as (S_1 ; S_2) and (R_1 ; S_2).

- **Cumulative:** In the cumulative context, all instances of each event type in the composite event are included in the composite event. Under this context, the composite event SE would include both S_2 and S_3 (R_1 ; <S_2, S_3> ; R_2) when it is detected at time 7.

In order to add a similar expressiveness to real-time logic, Liu, Mok and Konana (1998) extend their specification by introducing two events $E_s(TC)$ and $E_v(TC)$, which occur whenever the time constraint TC is satisfied or violated, respectively. Also, the function occ(e,i) is defined to be true if @(e,i) is defined and @(e,i) ≠ 0. Similarly, the function $occ_r(e,t,i)$ is defined to be true if @(e,t,i) is defined and @(e,t,i) ≠ 0. Given these extensions, the recent and chronicle contexts can be represented in real-time logic. Continuous and cumulative context can probably not be represented, and similarly there are expressions in real-time logic that cannot be expressed in Snoop. There are, thus, slight differences in the expressiveness of the two languages. Below, three different translations of the Snoop event (E1 ; E2) are shown as an example

i) $(E1 ; E2) \leftrightarrow E_s(occ(E2,i) \wedge @(E1,-1) < @(E2,i))$

ii) $(E1 ; E2) \leftrightarrow E_s(occ(E1,i) \wedge occ(E2,i) \wedge @(E1,i) < @(E2,i))$

iii) $(E1 ; E2) \leftrightarrow E_s(occ(E2,i) \wedge occ_r(E1, @(E2,i),0) \wedge @_r(E1, @(E2,i),0) \geq @(E2,i-1))$

In *(i)*, event (E1 ; E2) occurs whenever an instance of event E2 occurs provided any instance of event E1 has already occurred. This is essentially recent context, given that only one instance of event E1 occurs. In *(ii)*, event (E1 ; E2) occurs whenever the i:th instance of event E2 occurs, provided the i:th instance of E1 has already occurred. This is equivalent to chronicle context. In *(iii)*, event (E1 ; E2) occurs whenever an instance of event E2 occurs, provided an instance of event E1 has occurred at or after the previous occurrence of event E2. To our knowledge, there is no documented way to differ between *(ii)* and *(iii)* in Snoop.

2.4 Event monitoring with time constraint graphs

A constraint conjunction can be converted into a *constraint graph* (Chodrow, Jahanian and Donner, 1991). This kind of directed weighted graph can be used to monitor the constraints that it represents during the run-time of the monitor by instantiating the nodes in the graphs as events occur, and then checking for violated or satisfied constraints.

Each basic time constraint is converted into a pair of vertexes connected by a directed, weighted edge. As an example, consider a part of the timing constraint specification for the Millennium Falcon (Lucas, 1977): @(Start_hyperdrive,i)+4≥ @(Enter_hyperspace,i). This basic time constraint, which will be violated whenever the ship fails to enter hyperspace within four time units from the start of the hyperdrive engines (“it’s not fair!”), can be represented by the graph in figure 2.3.

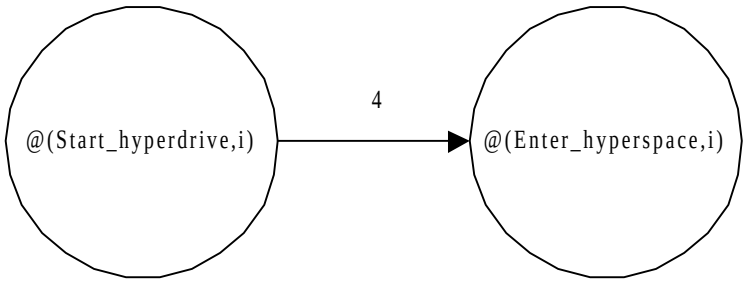

Figure 2.3: Constraint graph example 1

The weights on the edges in a constraint graph correspond to the integer values in the corresponding time constraints (the D value in T1 + D ≥ T2), and the vertices represent the @- or $@_r$-functions. A vertex that corresponds to a @-function is called an *absolute vertex*, and a vertex that represents a $@_r$-function is called a *relative vertex* (Mok and Liu, 1997b).

When an event instance occurs, the corresponding vertex in the constraint graph is said to be instantiated, and an occurrence time is associated with that vertex. If both vertices corresponding to the constraint T1 + D ≥ T2 are instantiated, the constraint can be checked for violation or satisfaction by evaluating the inequality.

As an extended example, the following constraint conjunction can be represented by the graph in figure 2.4:

$@(E1,i) - 6 \geq @(E2,i + 1) \wedge @(E1,i) + 5 \geq @_r(E3,@(E1,i),1) \wedge$

$@(E2,i + 1) + 9 \geq @_r(E3,@(E2,i + 1),1) \wedge 10 \geq @(E1,i)$

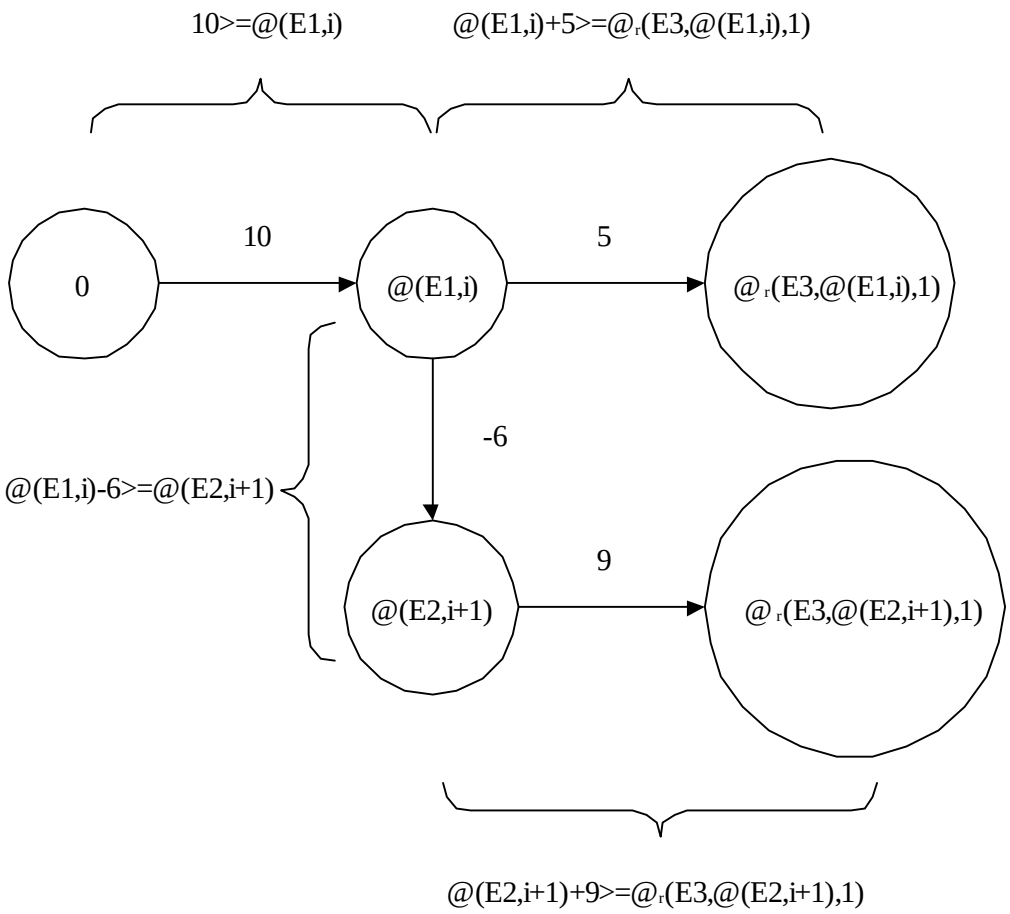

Figure 2.4: Constraint graph example 2

The constraint 10≥@(E1,i) is a special kind of basic time constraint, which represents that the occurrence time of the i:th event of type E1 should be less than or equal to 10. A special zero vertex is used to enable representation of constraints of this type.

Constraint graphs are a useful way of representing time constraints, since they facilitate the search for *implicit constraints*. Implicit constraints are constraints that are not explicitly defined, but that can be derived from the constraint specification. Consider the following constraint conjunction:

$$@(E1,i) + 1000 \geq @(E2,i) \wedge @(E2,i) - 999 \geq @(E3,i)$$

If the i:th instance of event E1 occurs at time 1 and none of the i:th instances of events E2 or E3 has occurred at time 2, the constraint conjunction above is violated. However, given only the constraints in the conjunction, this violation is not detected until both E2 and E3 has occurred, or at time 1000. The implicit constraint @(E1,i) + 1 ≥ @(E3,i), which can be derived from the constraint conjunction above, will be

violated at time 2. Thus, by finding and including implicit constraints in the constraint graphs, violations can be detected earlier.

2.5 Event monitoring objectives

Regardless of what an event monitor is used for, there are certain properties that are required. Mok and Liu (1997b) present three goals that should be achieved by the monitoring system; transparent monitoring, bounded detection latency, and minimum monitoring overhead. Out of these, transparent monitoring and bounded detection latency are both necessary requirements, while minimum monitoring overhead is a desirable property.

2.5.1 Transparent monitoring

In order to minimize the coupling between the monitoring system and the system being monitored, care should, according to Mok and Liu, be taken to separate the two systems as much as possible. Separating the systems implies that the amount of monitor-specific code in the monitored system should be minimized. Developers of system applications should not need to modify their code to allow their applications to be monitored.

2.5.2 Bounded detection latency

The most important aspect of an event monitor is that it has predictable resource requirements. It is crucial that events are detected within a bounded duration, and that there is a similar bound on the time required to detect that a time constraint has been satisfied or violated and react accordingly. Should no such bounds exist, the monitor loses much of its value, since it cannot be guaranteed that the monitor observes event occurrences in time.

2.5.3 Minimum monitoring overhead

As far as possible, the monitoring system should not affect the execution of the monitored system. This means that the monitoring process should use as little resources as possible, so that the monitored system is not hindered by having to compete with the monitor for processor time, memory, network bandwidth etc. A problem with monitoring a system is that a probe effect (Gait, 1985) can be introduced. The *probe effect* is the difference in behaivor between a monitored system and its non-monitored counterpart. To avoid this, when monitoring for testing and debugging purposes, the monitor must be left in the operational system (Schütz, 1994). Hence, the overhead of the monitor must be minimized.

The task of minimizing the monitoring overhead can be seen as maximizing the monitor's efficiency. When improvement of the monitor's efficiency is an issue, the efficiency and complexity of the algorithms used must be considered. According to Weiss (1999), an algorithm's *complexity* is a measurement of how fast the amount of resources needed for the algorithm increases with the number of input values. Generally, the less complex an algorithm is, the better it scales to large numbers of input values.

Algorithm complexity can be divided into *time complexity* and *space complexity*. Time complexity is a measure of how fast the time required for an algorithm increases as the number of input values is increased. Space complexity is similar, but is a measure of

the increase in the amount of resources (such as memory) that the algorithm needs during its execution.

Most of the time, the least complex algorithm for solving a problem is thus also the most efficient one. However, sometimes assumptions can be made about the environment where the algorithm is used that make a more complex algorithm the correct choice. Such an assumption could be an upper bound on the number of input values (for example, events in an event history) to the algorithm. An algorithm with a high complexity could be less time-consuming (more time-efficient) than one with a low complexity when the number of input values to be processed are lower than a given value. If we can assume that our system never requires more values than this to be processed, then we would be advised to use the more complex algorithm (or a hybrid of the two algorithms).

For example, assume that two algorithms, F and G, both take n input values. In figure 2.5, the functions $f(n)=5n$ and $g(n)=n^2$ represent the number of clock cycles needed for F and G, respectively, to produce a result. From the graphs we can see that the time required for F to complete clearly does not increase as rapidly as the time required for G as the number of input values is increased. Thus, F is less complex than G (in the time domain). However, if the number of input values in a system can be assumed to never exceed 5, G will be the most efficient algorithm for that particular system, since the execution time for G is lower than the execution time for F in all situations where the number of input values is lower than 5.

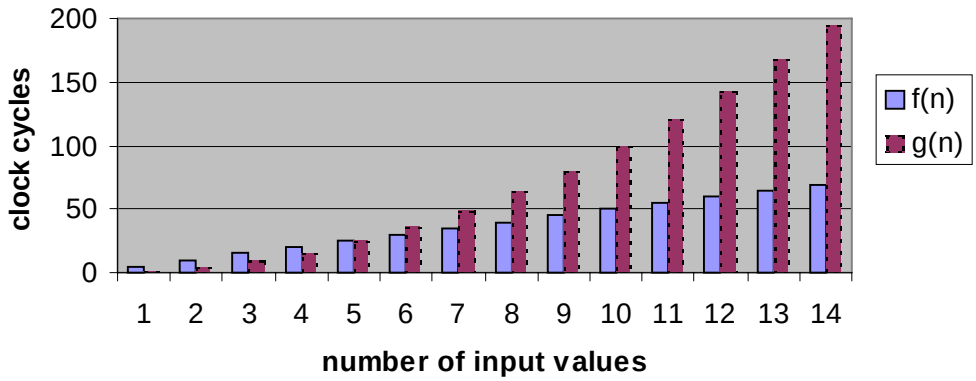

Figure 2.5: Complexity-efficiency graph

2.6 Algorithm optimization

As discussed in the previous section, the algorithms used by an event monitor must be efficient, especially in terms of computation time. In order to be efficient, the algorithms must be sufficiently *optimized*. Optimizing an algorithm means that an effort is made to minimize the amount of resources (in time or space) that the algorithm needs. Optimizing a program or algorithm can be facilitated by following certain guidelines. Webber (1992) presents a number of such guidelines, which he refers to as

the Principle of Least Computations. The essence of this principle is that a computation is unnecessary if

- the result of the computation is already known, or
- the result of the computation is not needed, neither as an output of the program nor as an input for a future computation

If the program already knows the result of a computation, then the computation can be eliminated. However, this implies that the result is stored, either in memory or on secondary storage. Thus, in decreasing the program's execution time we also increase the amount of resources it uses. Most of the time, however, processor time is a more limited resource than memory, and the trade-off is justified.

2.7 Existing work on constraint graph-based monitoring

An approach for monitoring of events on a uniprocessor system is described in Mok and Liu (1997b). In this paper, a way of specifying time constraints based on real-time logic is presented. These specification statements can be converted into constraint graphs, and Mok and Liu describes algorithms for *(i)* deriving implicit time constraints from constraint graphs, *(ii)* optimizing the graphs for efficient run-time checking of time constraints, and *(iii)* actually instantiating, updating and checking the graphs as events occur. These concepts are further discussed in chapter 4.

Mok and Liu's work is based on an approach originally developed by Chodrow, Jahanian and Donner (1991). However, Mok and Liu have improved the approach in several ways. Most importantly, the optimization of the constraint graphs done in Mok and Liu's method improves the time complexity of the constraint checking algorithms from $O(n^3)$ to $O(n)$, where n relates to the amount of vertexes in the constraint graph.

3 Problem description

This chapter explains and motivates the project focus, and describes the problems associated with creating a time constraint graph-based event monitoring design.

3.1 Purpose and project focus

The purpose of this project is to provide a flexible event monitor design that can serve as a basis for an implementation of Mok and Liu's event monitoring approach (1997b, 1998). The design focus is on issues that may have an influence on efficiency, and, in particular, on different methods for representing constraint graphs internally.

3.2 Motivation for efficient event monitoring

No matter for what purpose an event monitor is used, it is important that the monitor is efficient. As described in section 2.2.5 (on pages 6-7), the three main applications for event monitoring are testing, debugging, and fault tolerance. In all of these cases, monitoring efficiency is important. If the extra system load introduced with a monitor in a system is high, it may render a previously schedulable load unschedulable. In such a case, the event monitor is not sufficiently efficient, and as a result the monitored system is not timely.

3.3 Assumptions about monitoring and the monitoring environment

An event monitor can be seen as having an event-collecting part and a constraint-checking part. In this project, the part of the event monitor that handles the collecting of event information from the system, which is system-specific, is not considered. Further, it is assumed that event occurrences are detected instantly by the event monitor.

If the monitor is used in a distributed real-time system, it is also required and assumed that the messages (and thus the events) in the system are *stable* (Schneider 1994). In short, this means that all events are sent to and processed by the monitor in the order that they occurred. For simplicity, it is also assumed that all events in the network (regardless of which network node they are produced on) are detected instantly and have a correct timestamp based on a global time.

It is also assumed that the system provides sufficient resources for the event monitor to run. For example, there is enough memory to store event histories of the required size and the task load including the monitor is schedulable.

3.4 Using time constraint graphs for efficient event monitoring

As described in section 2.5, an event monitor should be transparent, i.e., there should be no need to introduce monitor-specific code into the monitored system, and predictable, i.e., there should be a bound on the monitor's resource requirement and efficient. The approach presented by Mok and Liu conforms well to these objectives. If all primitive events in the system can be detected by means of hardware (e.g., by hardware sensors or bus snooping) no monitor-specific code is needed in the monitored system. Thus, transparency can be achieved. Given the assumptions that *(i)* sufficient resources exist for the monitor to run, *(ii)* events are stable, and *(iii)* the time needed for event detection is bounded, then the event monitor is predictable. Compared to other, similar, methods (such as the approach presented by Chodrow,

Jahanian and Donner (1991)), it is also efficient in terms of algorithm complexity, since the time complexity of the algorithms used during run-time is only $O(n)$ with regard to the number of vertices in a given time constraint graph. There are, however, factors that may affect efficiency that are not covered by Mok and Liu, such as the event monitor's internal representation of the constraint graphs.

3.5 The data storage and representation problem

In an implementation of the event monitoring method presented by Mok and Liu (1997b) there are several places where object relations need to be represented. For example, the event monitor needs to maintain a list of all the constraint graphs that should be monitored. In an object-oriented design (which is used in this project), this could translate to an event monitor object keeping track of several constraint graph objects. The same can be said for the relation between constraint graphs and vertexes, as well as several other relations in the system.

The question is how these relations in general, and the constraint graphs in particular, should be represented in the implementation. The primary problem is what kind of data structure should be used. There are a number of options, each of which has certain advantages and disadvantages. The structure types evaluated in this project are dynamic arrays, static arrays, and linked lists.

4 Method

In this chapter, the approach taken to handle the issues presented in chapter 3 is presented. In particular, a design based on Mok and Liu's monitoring approach is described. Sections 4.1 and 4.2 give an overview of the central concepts in the design, while section 4.3 contains a more detailed description of the complete design.

4.1 Design overview

The design consists of two main parts – the parser and the monitor itself. The parser is used during the first phase of execution, which can be referred to as the parsing phase (see figure 4.1). During this phase, the user-specified event types are converted into an internal format. The parsing phase is followed by the setup phase, during which this internal format is loaded by the monitor. During the final phase, the run-time phase, the actual monitoring of the events is done. Also, certain events types can be added to or removed from the monitor during this phase.

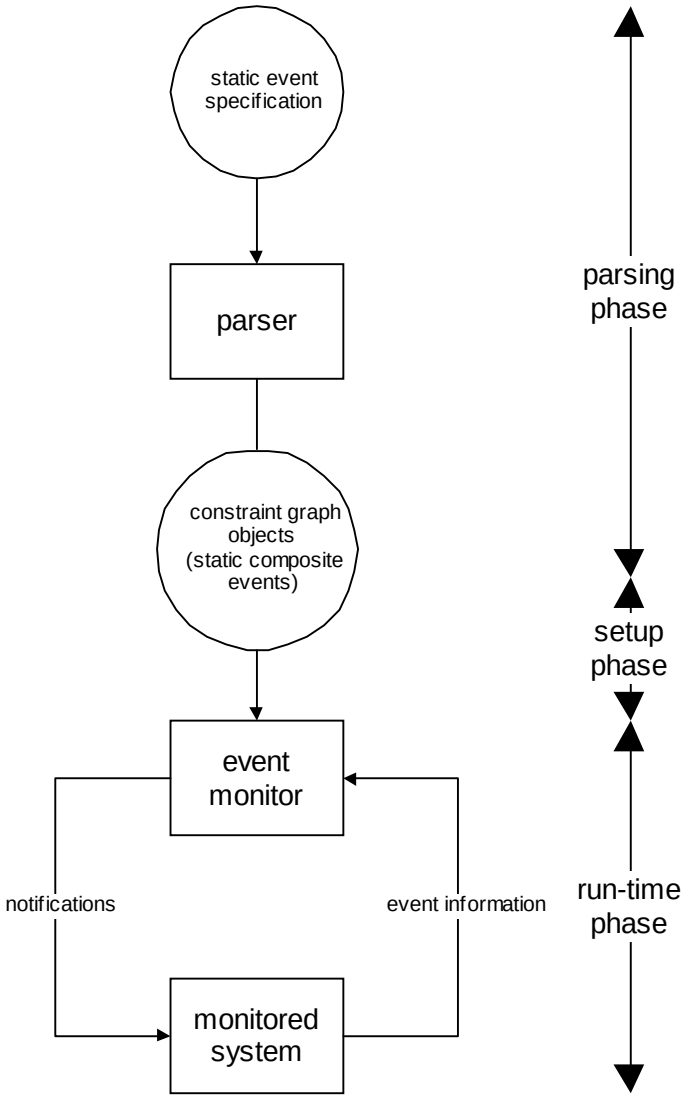

Figure 4.1: Monitor execution phases

4.1.1 Central object model

The design presented in this chapter focuses on the representation and handling of constraint graphs. Figure 4.2 shows the relationships between constraint graphs, event types, and the event monitor.

A single, permanent event monitor object exists during the entire monitoring session.

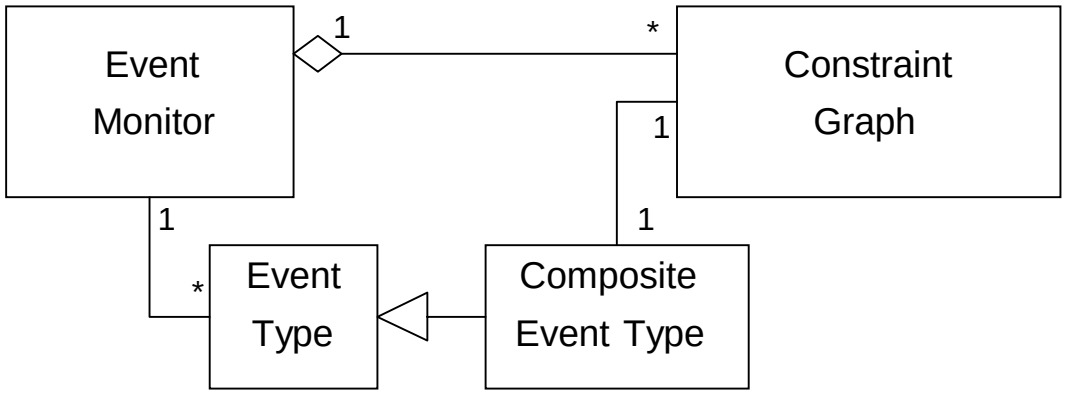

Figure 4.2: Central object model

This event monitor object maintains a number of event type objects, which can represent primitive or composite event types. Composite event type objects are associated with a number of constraint graphs that are used to determine when the composite event type has been detected. Instances of the event type, composite event type and constraint graph classes are used as the internal format of the time constraint specification.

The relation between the event monitor and the constraint graphs in the system is not strictly necessary, since it is a derived relation, but is useful in order to quickly find the relevant graphs when an event occurrence is signaled to the event monitor. Figure 4.3 shows a schematic view of the main collaborations of the central classes

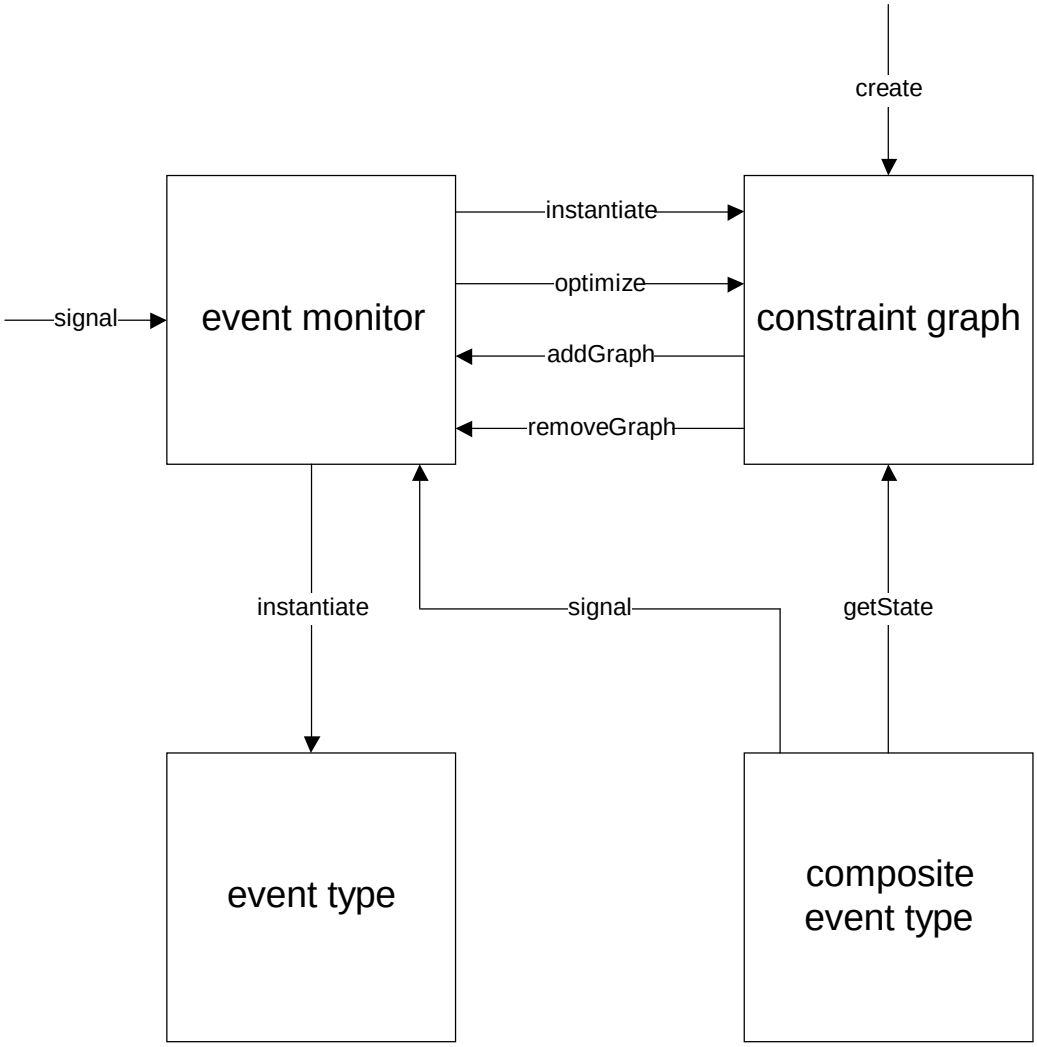

Figure 4.3: Collaborations of central classes

The event monitor object receives and processes event occurrence signals from the monitored system and from composite event type objects. When such a signal is received, information about the event occurrence is sent to the relevant event type and constraint graph objects via the instantiate interface methods.

When a constraint graph object has been created it adds itself to the event monitor object. If a constraint graph object is removed for any reason, it can also remove itself from the event monitor. The constraint graph interface also contains a getState function, which is used by composite event type objects to find out if the time constraints represented by the constraint graph has been satisfied or violated.

For a detailed description of the class interfaces in the model, see Appendix C.

4.2 Data structure types and object relations

As described in section 3.5, object relations can be represented by different types of data structures. The data structure types that are evaluated in this project are dynamic arrays, static arrays, and linked lists. The properties of each of these are:

- **Dynamic arrays** are flexible in size, since the number of slots in the array can be changed at run-time. This means that no unused memory is ever allocated, as can be the case with static arrays. However, the allocation and deallocation of memory that is necessary to maintain a dynamic array may lead to external memory fragmentation, especially if the arrays are large.
- **Static arrays** have a fixed size, which means that no new memory has to be allocated at run-time. The drawback is that the maximum size of the arrays must be predicted at compile time, and if these assumptions are pessimistic a lot of memory is wasted.
- **Linked lists** are more space-efficient than static arrays with many unused slots. They are, however, less space-efficient than dynamic arrays, since a pointer to each data entry must also be stored. However, since it is not necessary to store all entries in a linked list linearly in memory (as is the case with arrays), external memory fragmentation is not a concern.

  Linked lists allow data to be inserted and removed efficiently with relatively little memory management. However, if a linked list is to be copied, each data entry must be copied individually, resulting in an amount of subroutine calls equal to the number of data entries. If an array is used, the linear storage of data means that all entries can be copied as a single memory block.

4.3 Extended design

In this section, an extension of the basic design described in sections 4.1 and 4.2 is presented.

4.3.1 Event model

For the purposes of this project, events types are separated into *static events types* and *dynamic event types*. Static events types are events types that are specified before the monitor execution is started, and dynamic event types are events types that can be added or removed at run-time. The table in figure 4.4 shows the relation between event types and their representation and specification method.

Event type	Specification type	Internal representation
primitive	static only	instance of class Etype
composite	static or dynamic	constraint graph

Table 4.4: Specification of event types

As can be seen in the table, primitive event types cannot be specified dynamically. It is assumed that all necessary primitive event types are already identified and exist in the system when the monitor is started. This assumption is reasonable in systems that do not allow new elements (e.g., new hardware) to be added to the system during run-time. In a system where such dynamic additions are allowed, new primitive events that are related to the new elements would have to be added to the monitor dynamically. If the presented design is used in such a system, the monitor would have to be restarted each time the set of primitive event types changes.

4.3.2 Event parameter model

In the design presented here, event parameters can only be specified for primitive event types. Composite events are assumed not to have any parameters other than the default parameters (occurrence time, event type) and those that can be derived from its constituents.

4.3.3 The parser and the event specification language

The code for the parser was created using the UNIX programs lex (for tokenizing/keyword recognition) and yacc (for parsing). The parser's task is to read a specification and generate constraint graph objects (instances of the constraint graph class). These objects are related to one composite event each, and are loaded by the monitor either during the setup phase (for static event types) or during the run-time phase.

Specification of static composite event types

Composite event types are specified using a simple specification language. Figure 4.5 shows the specifications of three different composite events.

a)

```
event Hyperfail is violation of
[
  @(Start_hyperdrive,i)+4>=@(Enter_hyperspace,i)
]
```

b)

```
event TransactionMetDeadline is satisfaction of
[
  @(Start_transaction,i)+5>=@(Abort_transaction,i) ||
  @(Start_transaction,i)+5>=@(Commit_transaction,i)
]
```

c)

```
event Zebrafail is violation of
[
  @(Walk_light_off,i)-10>=@(Walk_light_on,i) &&
  @(Walk_light_on,i)+40>=@(Walk_light_on,i+1)
]
```

Figure 4.5: Composite event type examples

4.5 (a) shows the specification of an event type based on the constraint graph example in figure 2.3 (on page 11). That is, it represents one of the constraints that can be posed on a spaceship hyperdrive. The composite event type in 4.5 (b) could be used to specify that a transaction has met its deadline if it is either aborted or committed within 5 time units. Finally, the composite event type in 4.5 (c) is instantiated if the walk light in a zebra crossing is lit for a duration of less than 10 time units, or if there is a period of more than 40 time units between two occurrences of the event Walk_light_on.

“Hyperfail”, “TransactionMetDeadline”, and “Zebrafail” are examples of event names. The event name is a user-specified string that must start with a capital letter and is used as the name of the composite event type. The satisfaction and violation keywords are used to specify whether the event should be signaled on satisfaction or violation, respectively, of the specified time constraint graph. The basic time constraints specified between the square brackets must be written in disjunctive normal form (see the definition of time constraints in section 2.3), separated by “&&” (AND) or “||” (OR).

See appendix A for a formal definition of the language in extended Backus-Naur Form (ISO, 1996).

Specification of dynamic composite event types

Dynamic composite event types are specified using the same language as static composite event types, and can thus be parsed using the same parser. The difference is that the specification of static composite event types is parsed before the monitor is started, while addition of dynamic composite event types are requested by the user, as shown in figure 4.6.

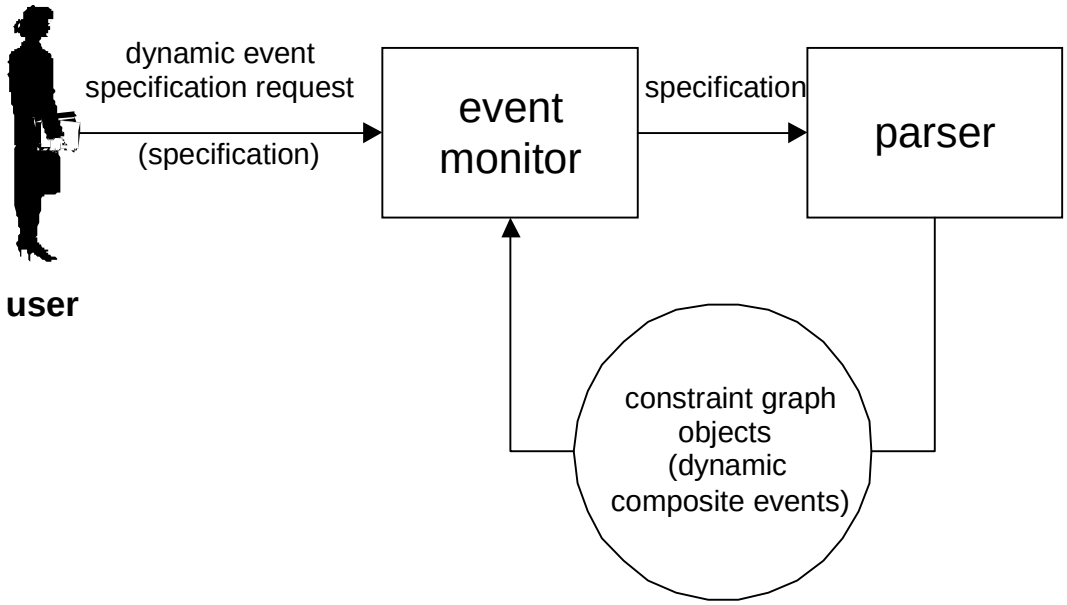

Figure 4.6: Dynamic event specification

When dynamic event types are added, the specification is sent to the event monitor. The monitor invokes the parser, which generates time constraint graphs and composite

event type objects from the specifications in the file and adds them to the monitor.

Specification of primitive event types

To specify a primitive event type, an object of the event type class is created. Exactly how this is done depends on the programming language used. By modifying the event type class, events types can be modified to contain more parameters than the default parameters. A possible way of specifying different parameter sets for different event types is to use the event type class as a base class for multiple event type subclasses, as shown in figure 4.7.

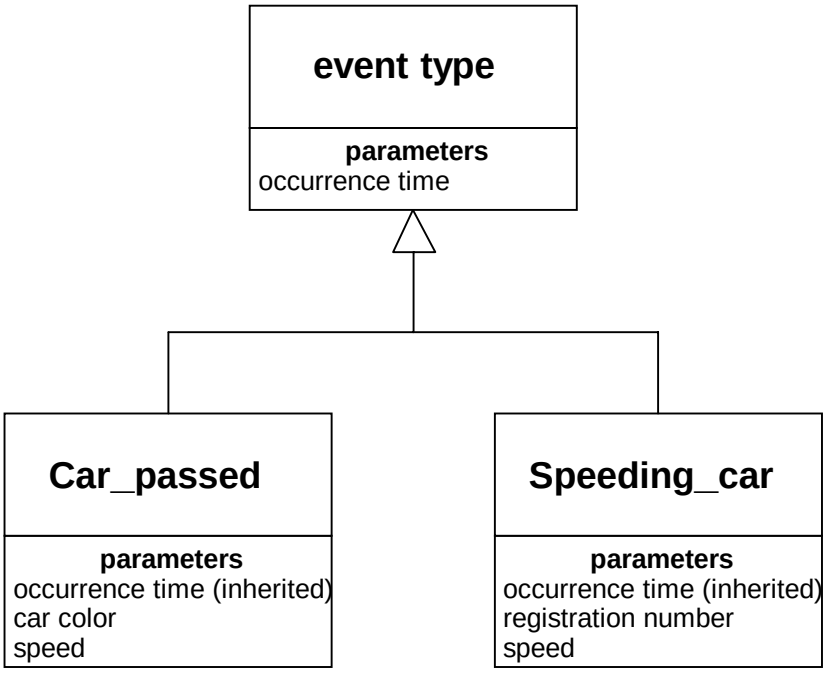

Figure 4.7: Event parameter inheritance

In this figure, the event type class is the superclass of two event type subclasses that could be used in a traffic surveillance system. The superclass have only one parameter (the occurrence time), which is inherited by the subclasses. However, the subclasses also add certain parameters of their own – car color and speed in the Car_passed subclass, and registration number and speed in the Speeding_car subclass. All parameters of an event type subclass are recorded in the event history of that class.

4.3.4 Complete object model

The object model for the implementation is presented in figure 4.8.

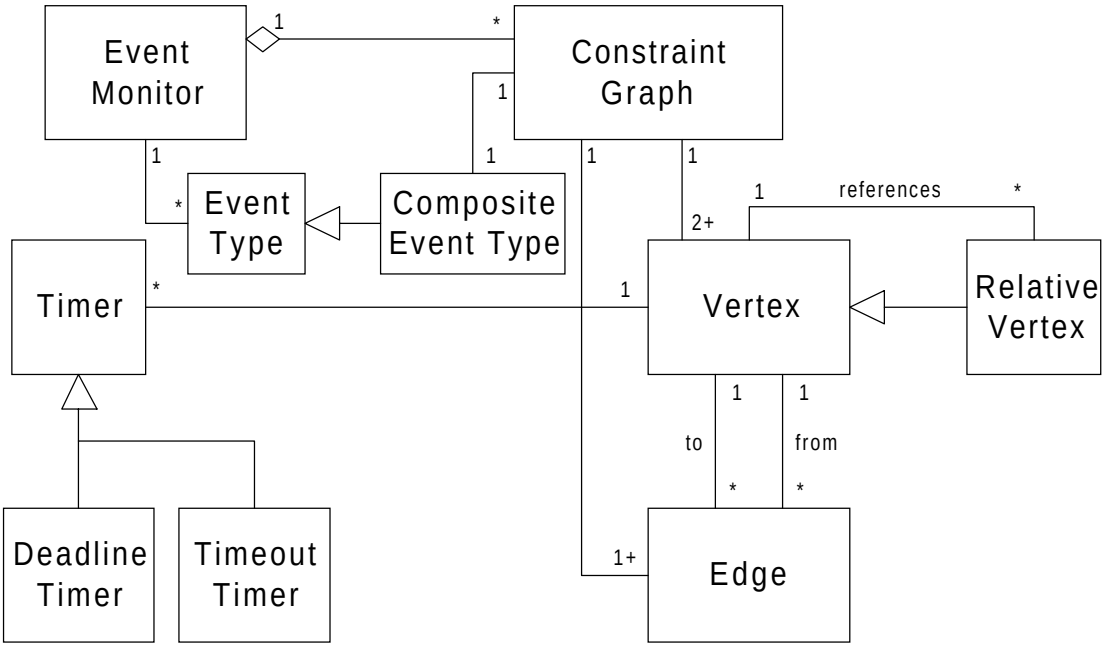

Figure 4.8: Complete object model

A short summary of the classes in the model follows:

Event monitor

An event monitor object is the 'core' of the implementation. It contains a number of constraint graphs (the monitored graphs) and a table that lists which graphs are associated with a certain event type. This table is used when an event instance occurs in order to determine which constraint graphs should be notified of the occurrence.

Event type

Each direct instance of the event type class represents a primitive event type. It contains an event history which stores the occurrence time and parameters of all instances of this event type.

Composite event type

Each composite event type object is associated with a number of constraint graphs. Constraint graphs represent the signaling rules for the composite event. The composite event type class is a subclass of the event type class, and inherits the properties of that class – most importantly the event histories.

Constraint graph

A constraint graph object contains a number of vertex objects and a number of edge objects. Each connected pair of vertices represents a basic time constraint (see section 2.4 on page 10).

Vertex

A vertex object contains the name of the event type associated with the vertex and an index. The index, which is an arithmetic expression on the form $a*i+b$ (a and b integers, i a variable), is used to determine which instance of the associated event type should be used for instantiation of the vertex.

Relative vertex

A relative vertex object has the same properties as a vertex object, but adds an association to another vertex object (the reference vertex of the relative vertex).

Edge

Each edge object is associated to two vertex objects, which define the start and end of the edge. It also contains the weight of the edge, which represents the deadline or timeout time for a time constraint (the D value in T1 + D ≥ T2).

Timer

Timer is an abstract superclass of the timeout and deadline timer classes. Each timer object, regardless of type, contains an association to the vertex that the timer notifies whenever a timeout is detected or a deadline is reached. Each vertex controls the creation and termination of all timers that are related to the instantiation of the vertex. Timers are often needed to ensure early detection of constraint violations (Mok and Liu, 1997a).

Assume that a time constraint on the form $T_1 + D \geq T_2$ is monitored, and that the vertices V_1 and V_2 are the corresponding vertices to T_1 and T_2, respectively. If V_1 and V_2 are both instantiated, then the constraint can be evaluated. However, if only one vertex is instantiated, then a timer might be necessary to ensure detection of a constraint violation. The table in figure 4.9 shows the timers needed in a given situation (Mok and Liu, 1997a).

State of V_1	State of V_2	Value of D	State of constraint
not instantiated	not instantiated	irrelevant	cannot be evaluated
instantiated	not instantiated	positive	deadline timer needed
instantiated	not instantiated	negative	always violated
not instantiated	instantiated	positive	always satisfied
not instantiated	instantiated	negative	timeout timer needed
instantiated	instantiated	irrelevant	can be evaluated

Table 4.9: Required timers (Mok and Liu, 1997a)

Deadline timer

An instance of a deadline timer object is created whenever a constraint implies a deadline for the instantiation of a vertex once the other vertex in the constraint has been instantiated. If the timed vertex is not instantiated before the timer is finished, the associated constraint has been violated.

Timeout timer

A timeout timer is used when a constraint specifies a necessary delay between the instantiations of two vertexes. If the timed vertex is instantiated before the timeout timer has finished, a violation has occurred.

4.3.6 Program flow

The event monitor is idle as long as there are no event occurrences in the system, but becomes active whenever a monitored event occurs. Figures 4.10 to 4.12 show

schematically what happens when the monitored system signals an event occurrence to the event monitor.

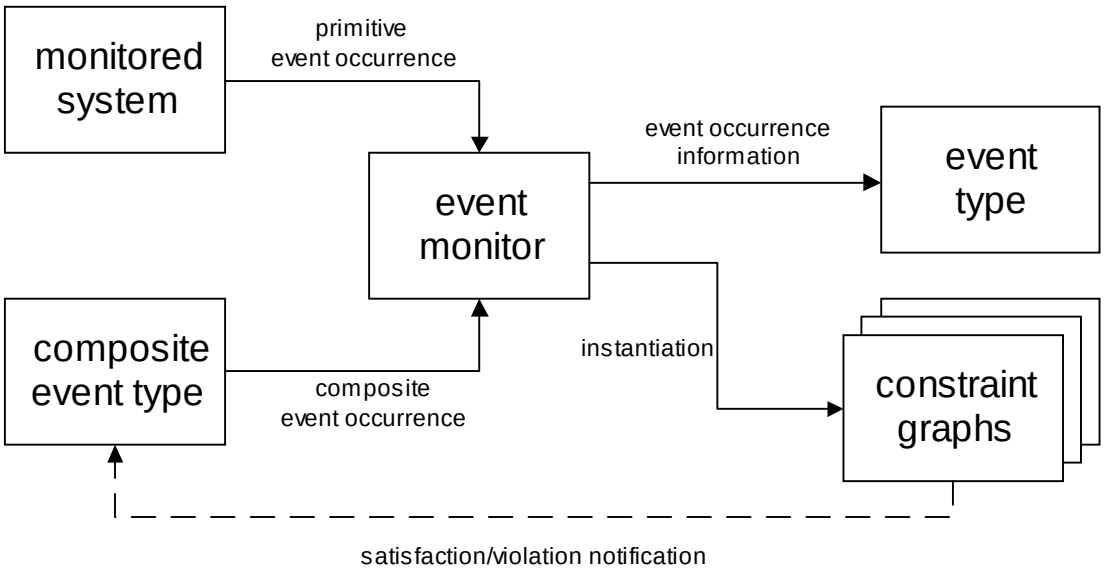

Figure 4.10: Overview of program flow

When the event monitor receives an event occurrence signal from the monitored system or from a composite event type, it immediately forwards this information to the relevant constraint graphs and the instantiated event type. The event type records the parameters of the instance in its history. Since an event history has a bounded size, this could cause the first history entry to be flushed from the history.

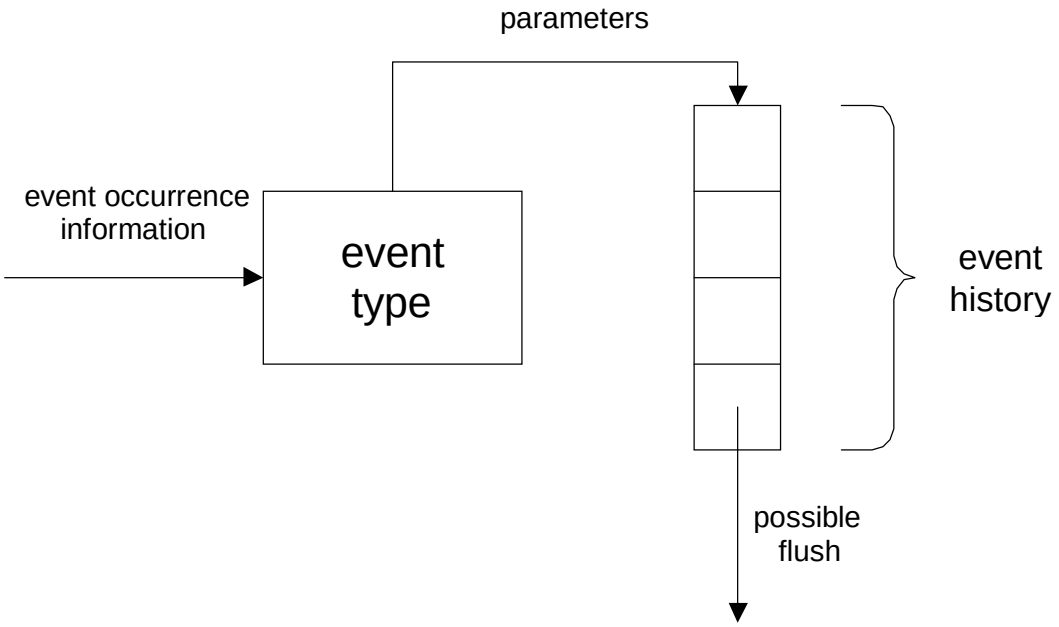

Figure 4.11: Instantiation of an event type

When a constraint graph becomes notified of an event instance, it searches its set of

vertices, and instantiates any matching vertex to the occurrence time of the event instance. If the graph is satisfied or violated as a result of the instantiation, this is signaled to the composite event that the graph is associated to (see figure 4.12).

If this causes the composite event to occur, an event instance of the composite event type is generated, and signaled to the event monitor. The monitor notifies the monitored system of the event occurrence, and the cycle starts over.

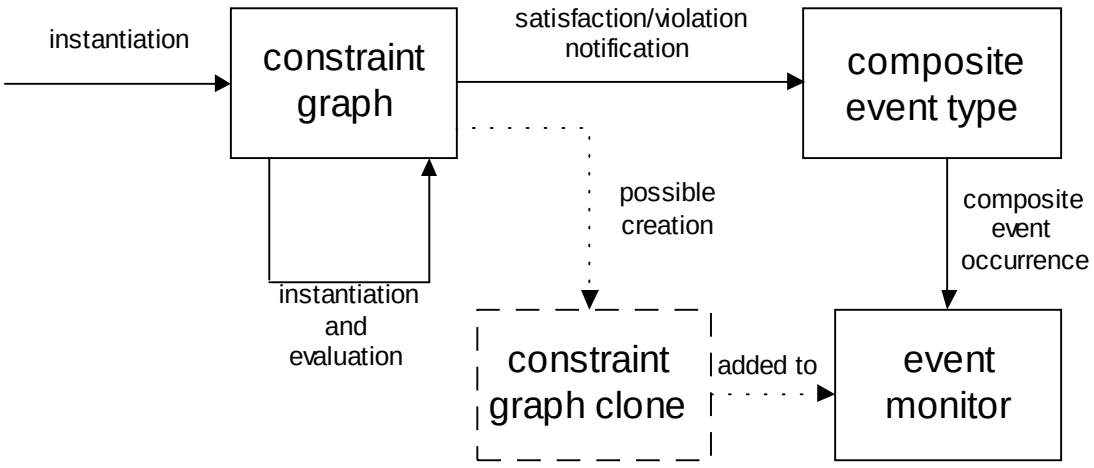

Figure 4.12: Instantiation of a constraint graph

Note that some vertices in the graph may contain arithmetic expressions (instead of numerical values) for their index. That is, they may relate to occurrence functions on the form @(Eventname,*a***i*+*b*). If such a vertex with a matching event name is found when the graph is being instantiated, the monitor tries to find an integer value for *i* that satisfies the equation *a***i*+*b* = event occurrence index. This is done by setting *i*=(event occurrence index - *b*) / *a*.

If a value for *i* is found, the constraint graph creates a clone of itself. This *clone* is a copy of the original constraint graph (and is thus associated with the same composite event type as the original), but all arithmetic index expressions in the cloned vertexes are replaced with their matching numerical values for the computed *i* value. The constraint graph clone object is added to the monitor, and functions as a regular constraint graph, with the exception that once the clone has been satisfied or violated, it is removed from the system. Consider the following example:

The composite event type ExampleEvent is defined as

```
event ExampleEvent is satisfaction of

[
   @(E3,3*i-4)+4>=@(E4,i) &&
   @(E3,3*i-4)-3>=@(E1,2) &&
   @(E1,2)+5>=@(E2,2*i)
]
```

This event type is represented by the graph in figure 4.13.

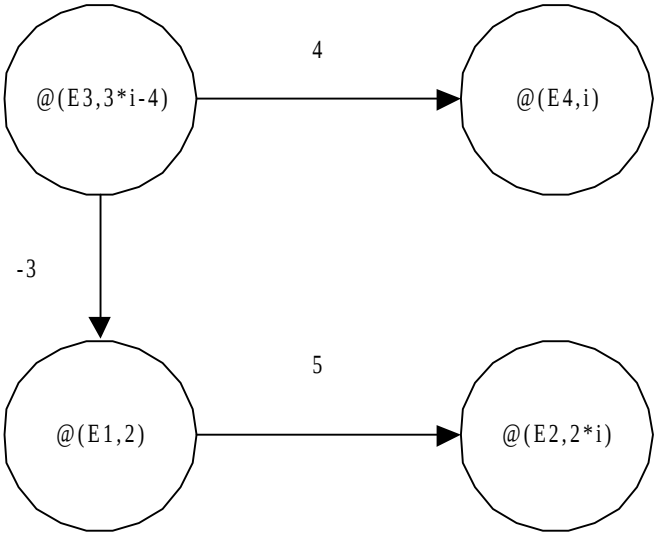

Figure 4.13: Constraint graph for ExampleEvent

Assume that the second occurrence of the event type E3 is signaled to the event monitor at time 9. The monitor forwards the information to the constraint graph, whose set of vertices is searched for a vertex that matches @(E3,2). No such vertex is found, but by setting *i*=2, @(E3,3**i*-4) would be equivalent to @(E3,2), since 3*2-4=2. Thus, the graph is cloned, and *i* is replaced with 2 in all vertexes in the cloned graph. The resulting clone is shown in figure 4.14.

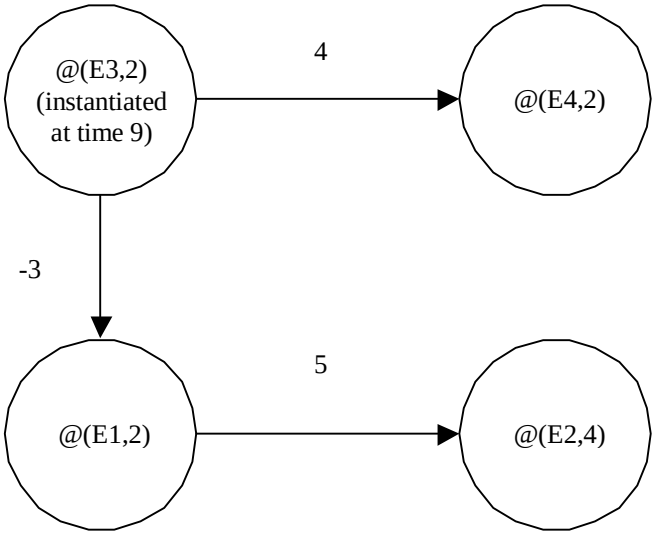

Figure 4.14: Constraint graph for ExampleEvent (after instantiation)

Note, that the vertex that is now corresponding to @(E3,2) is immediately instantiated, since it matches the event occurrence that triggered the cloning.

4.3.7 Algorithms

Three algorithms presented by Mok and Liu (1997b) are used in the design presented here. The algorithms and their respective functions are described below. For a complete pseudo-code description of the algorithms, see Mok and Liu (1997b).

Graph compilation

The graph compilation algorithm is used during the setup/parsing phase to remove all unnecessary constraints from a constraint graph, and also to find the shortest path between all pairs of nodes that correspond to a necessary constraint.

Constraint satisfiability check

Checks the state of a given time constraint in a constraint graph. It detects whether the time constraint is violated or satisfied, and can also determine if a deadline or timeout timer is needed in order to detect violation/satisfaction of the time constraint as early as possible. The constraint satisfiability check algorithm is used by the constraint graph satisfiability check algorithm (described below).

Constraint graph satisfiability check

Checks the satisfiability of a constraint graph. The constraint graph satisfiability check algorithm is used whenever a vertex in the graph becomes instantiated. It calls the constraint satisfiability check and update paths algorithms as necessary.

All of these algorithms are part of the constraint graph class.

5 Results

This chapter covers the results of test runs of an implementations of the design described in chapter 4. Initially, identified bottlenecks of the design are presented. Finally, the strengths and weaknesses of different methods for data storage and representation are discussed. Further, the circumstances that could make either of the methods more suitable are identified. The measured benchmarks are also presented.

5.1 Test approach

In order to test the efficiency of the different storage and representation methods presented in section 4.2, an example implementation in C++ (using Sun's C compiler version 3.0.1) of a primitive time constraint-based monitor has been constructed. In order to simplify configuration of the monitor, all object relations of variable cardinality were implemented using two template-based classes (called Chain and ChainIterator), which together give a flexible representation of a set. By changing the implementation of these classes, the efficiency of different structure types can easily be tested.

The Chain class

An object of the Chain class maintains a set of a specified data type. The interface of the class is described in Appendix C.

The ChainIterator class

Each instance of the ChainIterator class corresponds to a Chain object. The ChainIterator object can be used to iterate through all the Chain elements, and to extract a value from a certain position in the Chain. The ChainIterator can be seen as containing a pointer to a value in the corresponding Chain. This pointer can be moved by the member functions of the ChainIterator class. The ChainIterator interface is defined in Appendix C.

Three different implementations of the Chain and ChainIterator classes were constructed, one for each of the data structure types dynamic array, static array, and linked list.

5.2 Benchmarking method and operating system considerations

All tests presented in this chapter were conducted by timing the execution of test runs on a Sun Workstation Ultra-4 with an UltraSPARC 250MHz processor running the Solaris 2.6 operating system. However, very few operating system specific services are used by the event monitor design, mostly because of the fact that the monitor does not incorporate collecting of events from the system. The required services are message passing between processes (to communicate event occurrences from the monitored system to the monitor and notifications from the monitor to the monitored system), and system timers (used for the deadline and timeout timers). The event monitor can be implemented on any operating system that can provide these features.

5.3 Description of simulations

Three simulations were run in order to study the impact of three different factors, which had been identified as possible bottlenecks. These factors were *(i)* the number of constraint graphs in the system, *(ii)* the size of the constraint graphs (in number of constraints) and *(iii)* cloning vs non-cloning graph instantiations. In all of the simulations, the average time needed for an event instance (computed by dividing the total time needed for five test runs of 1000 event instances by 5*1000) was studied. The impact of three different factors (which had been identified as possible bottlenecks) was studied, each of which served as the focus for one of the simulations. The event specifications and main programs used for conducting the first two simulations were generated by two C++ programs, which can be found in Appendix B, part 3.

The first program, maingenerator.cc (listed on page 124), creates a main program for the simulation, based on the number of primitive events and event instantiations specified by the user. The resulting main program creates primitive event objects, and contains a loop that raises the specified number of primitive events. The second program, specgenerator.cc (listed on page 125), creates the file containing the composite event specifications. The user specifies the number of composite events to be generated, and also the size of the composite events.

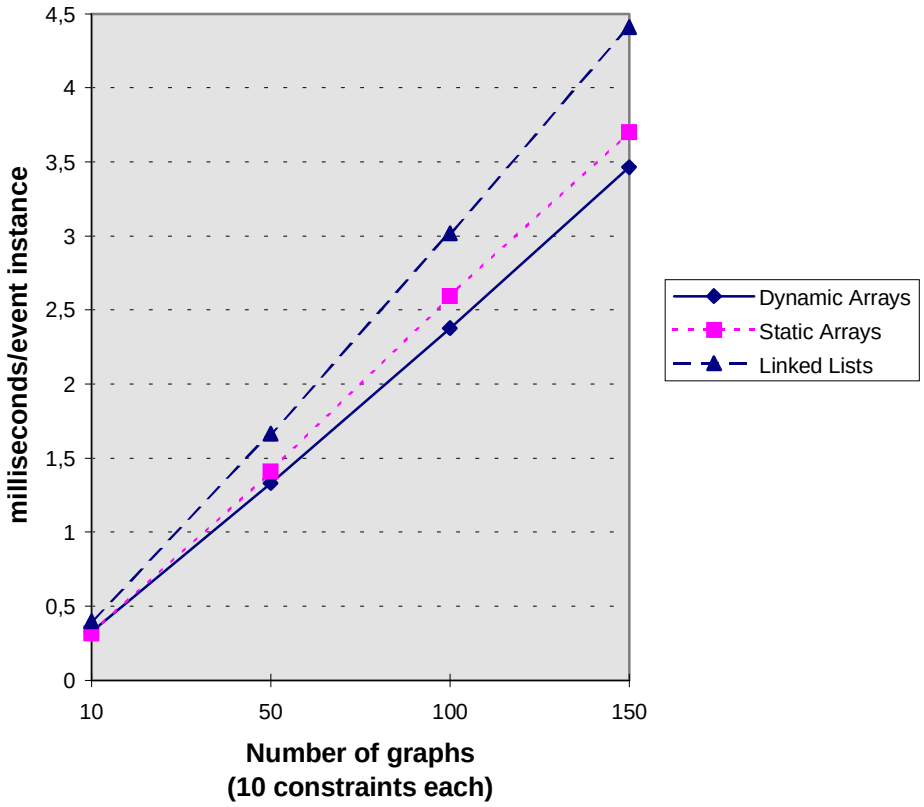

Figure 5.1: Results of simulation 1

5.3.1 Simulation 1 - number of constraint graphs

In the first simulation, the number of constraint graphs in the system was varied. The diagram in figure 5.1 shows the time needed for each event instantiation in four systems which contain 10, 50, 100, and 150 graphs, respectively, with each graph containing 10 constraints.

As can be seen in the diagram, the number of composite event types in the system has a considerable impact on the time needed for an event instantiation. This could probably be related to the inefficient search method that is currently used to find the graphs that should be notified of a particular event instance. Measuring the time used by the different functions in the program (using the gprof command in UNIX) confirms this, and as can be seen in figure 5.2, about 50% of the monitoring run-time is spent on this kind of search. By using a more efficient search method, this aspect of the monitor could be much improved (see section 5.6).

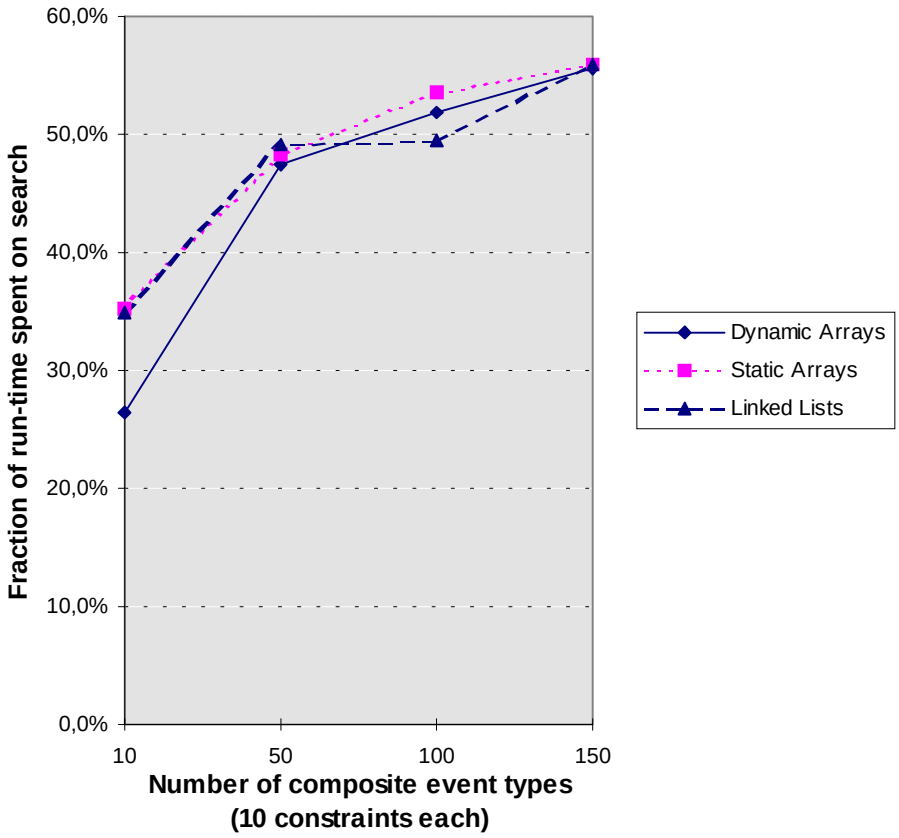

Figure 5.2: Search algorithm time usage

5.3.2 Simulation 2 - size of constraint graphs

In the second simulation, we varied the number of constraints in the monitored graphs. While this obviously affects the time needed for an event instantiation, the low time complexity ($O(n)$) of the algorithms used for checking the constraints (Mok and Liu, 1997b) ensures that the presented approach handles large graphs relatively well. The diagram in figure 5.3 shows the time needed for each event instantiation in a system

with 10 monitored graphs. Four test runs were made for each of the data structure types, with the graphs containing 10, 50, 100, and 150 constraints, respectively.

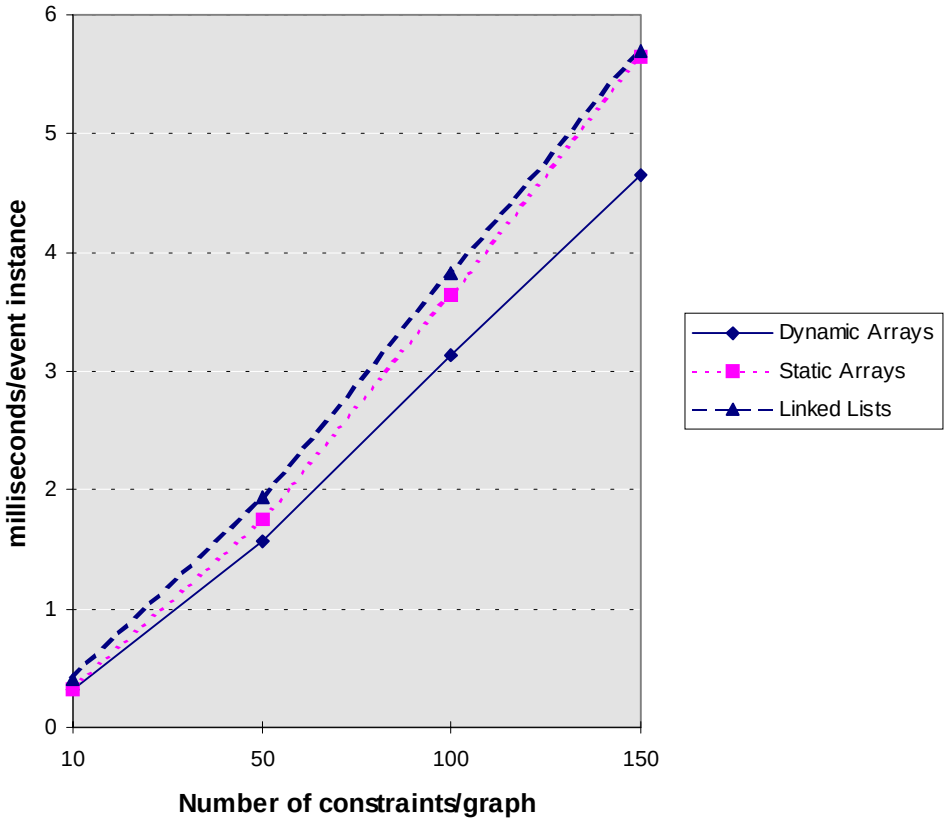

Figure 5.3: Results of simulation 2

As can be seen in figure 5.3, the time needed for instantiation of a graph increases almost linearly when the graph size is increased.

It is necessary to bear in mind that graphs, which represent parts of composite events, will most likely never contain as many as 50 or 100 constraints, since this would make the semantics of the composite events of which they are a part incredibly complex. It is, however, very likely that a system would contain many (100 - 150) composite events. This means that it is more important to ensure that an implementation can handle large amounts of graphs, than it is to support graphs that contain a large number of constraints.

5.3.3 Simulation 3 - graph cloning

The aspect that affects the event instantiation time the most, however, is the number of graph clonings that have to be done, as is shown by simulation 3. It is not hard to see why graph cloning is time consuming. The creation of a graph clone means that a new constraint graph object has to be created, all data in the cloned graph must be copied to the new graph, the vertices in the new graph must be traversed to replace all arithmetic indices with absolute values, the new graph must be instantiated and must finally be added to the monitor's list of monitored graphs. Figure 5.4 shows for each of the tested data structure types the time needed for an event instantiation that results in

a graph cloning, and also for an instantiation where no clone is created. Both tests were done in a system containing a single graph with a single constraint (the specifications used can be found in appendix B (part 4, page 126). In the test where graph clones were created, the search aspect was eliminated by removing the clones as soon as they had been created and added. Thus, the system never contained more than one graph. As can be seen, the time needed for a graph cloning is 3-4 times larger than the time needed for the same kind of instantiation without graph cloning.

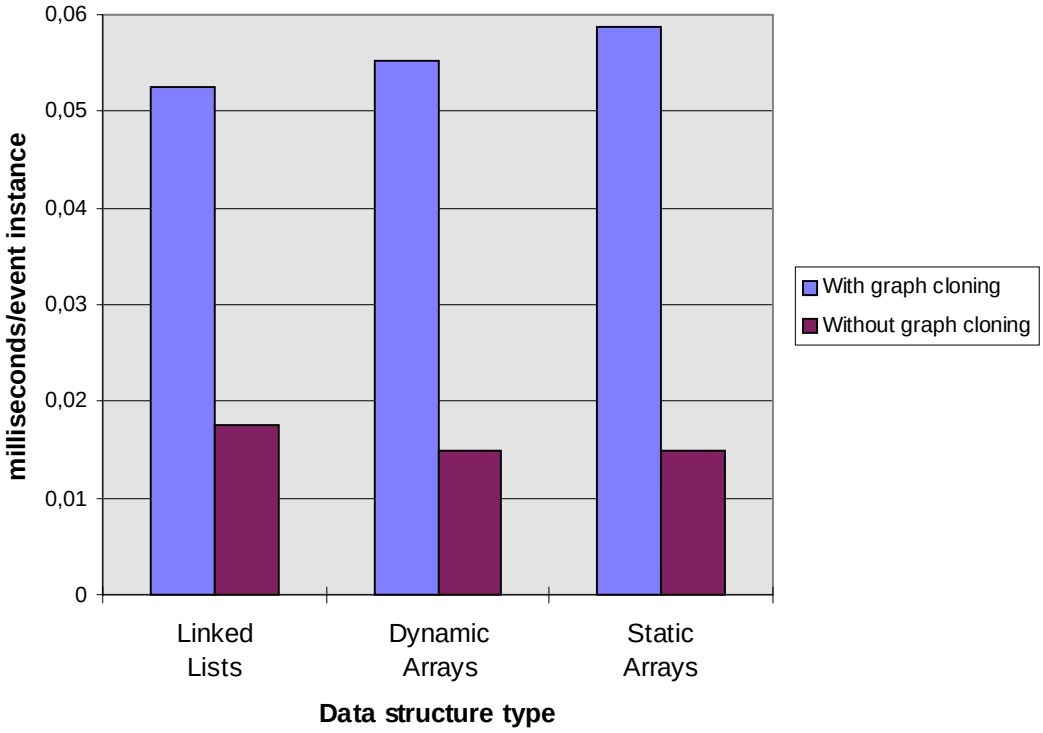

Figure 5.4: Results of simulation 3

5.4 Effect of different strategies for data storage and representation

As can be seen in figures 5.1, 5.3, and 5.4, the time needed for certain operations in the system varies noticeably depending on which data structure type was used during the test run. Although dynamic arrays consistently perform well, there are several factors specific to each data structure type that must be considered when analyzing the measured values. A discussion of each of the data structure types follows.

Linked lists

During the test runs, linked lists proved not to scale well to large numbers of graphs, nor to graphs that contain a large amount of constraints. It was, however, the most time-efficient data type for graph cloning. This can most probably be derived from the fact that while addition and removal of data entries can be done quickly with a linked list, operations that search or iterate through the list are done slowly due to the fact that the data entries are not lined up in memory. Since search operations are much more common than add- and remove operations in the monitor implementation, an inefficient search method is a serious disadvantage. Also, the linked list implementation

used for the test runs used an unnecessarily inefficient method for adding data entries (see section 5.5). Improving this would probably increase the performance of the linked lists noticeably.

Dynamic arrays

Dynamic arrays performed well in all the conducted tests. While add and remove operations are performed slower on dynamic arrays than on linked lists, the fact that arrays are stored linearly in memory make iterations through arrays efficient. Also, because of the dynamic nature of the arrays no memory is wasted, although external fragmentation may be an issue.

Static arrays

In the tests, static arrays generally performed worse than dynamic arrays. As with linked lists, modification of a static array by adding or removing data is performed more efficiently than performing the same operations on a dynamic array. However, since the amount of memory needed for a static array must be determined upon creation of the array, the amount of memory that the array uses must be estimated pessimistically. In the implementation used for the test runs, all static arrays in the system had the same upper bound on their size. This simplification can lead to high memory requirements and a huge memory waste if this bound must be set high because of the requirements of a particular structure. Such memory requirements may lead to unnecessary overhead if the memory allocation is slowed down by, e.g., fragmented memory. This kind of overhead may account for the differences between the test runs of dynamic and static arrays.

5.5 Implementation issues

There are some aspects of the event monitor implementation used for the tests that could be optimized in order to improve some of the test results. These optimizations are presented below:

- Each time an event occurrence signal is received by the monitor, the monitor must find all graphs that should be notified of the event occurrence. Currently, the relevant graphs are found by searching an unsorted set that connects each event type name with the graphs that contain vertices that correspond to events of that type. This set is updated each time a constraint graph is added to or removed from the monitor. Searching this set linearly is very inefficient, and the search time can be improved by replacing the current method with, for example, a search algorithm based on hash tables.
- Each time a data entry is to be added to a linked list, the list searches for its end node by iterating through all its nodes. The list then adds a node with the new data entry to the end of the list. If a pointer to the last node was maintained by the list, the addition of a data entry to the list could be made more efficient.
- Instead of using the same value for the maximum size of all static arrays in the system, each static array should have its own size bound. This way, much less memory would be wasted, which would probably improve the efficiency of static arrays.

5.6 Discussion of results

As the test results show, dynamic arrays should be the first structure type that is

considered when creating an implementation of a graph-based time constraint monitor. Dynamic arrays are flexible, since no estimation of their memory requirements have to be done, and are (as has been shown by the tests) efficient enough to handle most situations.

There are, however, a few scenarios in which one of the other data structure types could be considered. For example, if a system's memory is fragmented, large memory blocks such as those needed by arrays (of any type) may be difficult to find and allocate. In such a situation, a linked list may be more efficient than an array, since a linked list functions equally well in fragmented and non-fragmented memory because of its pointer-based nature.

Also, if the system requires only small data structures, and if a bound on the required size of these structures can be easily determined, static arrays may be more efficient than dynamic, since memory deallocation and allocation is not needed in maintaining a static array.

5.7 Related work

A constraint graph-based application called JECTOR (Java Event Correlator) is briefly discussed in Liu, Mok and Yang (1999). The approach taken in creating JECTOR is similar to the one taken in this project. An event specification language, JESL, is defined, and the JECTOR event monitor has a parsing component that functions in a way analogous to the event parser presented here.

While JECTOR was created to show how a constraint based event monitoring approach could be used in a real-world application, this project focuses on efficient implementation of the actual monitor. The presented event monitor model was designed to be easily modified in order to quickly evaluate different implementation configurations.

A task that has been shown to be a suitable application for JECTOR is to facilitate the identification of network problems in a network management system. Often, the failure of a single network object in such a system could cause large amounts of seemingly unrelated event occurrences (error messages). JECTOR can be used to analyze the event occurrences and aid the system administrator in finding the cause of the problem.

6 Conclusions

This chapter contains a summary of the dissertation, a presentation of the main contributions that have been made in this project, and also a summary of the work that has yet to be done.

6.1 Summary

This project focused on efficient implemenation of a graph-based time constraint monitor for real-time systems. The main problem is to evaluate different data structure types that can be used for a monitor's internal representation of constraint graphs.

To address these problems, an event monitor design was constructed which allows the data structure type used to be easily changed. Based on this, an event monitor prototype was created. A series of benchmarking tests were run in order to find bottlenecks in the design, and to evaluate the performance of implementations using different types of data structures.

The tests implied that the main bottleneck is the cloning of graphs (see section 4.3.6). Also, the benchmarks showed that dynamic arrays are generally the most time- (and space-) efficent way of representing a constraint graph. However, the difference in performance between the different data structure types is small enough to consider static arrays or linked lists in systems with properties that make any of these types more suitable. For example, a system where the data structures are small during the system's entire lifetime. In such a system, static arrays would probably outperform dynamic arrays.

6.2 Contributions

The following contributions have been identified in this project:

- A flexible, object-oriented implementation model for a time constraint graph-based event monitor design. The presented design has clearly separated parts, each with a well-defined interface, which allows modifications and configurations of the design to be done easily. In particular, the data structure type used for representing the time constraint graphs can easily be changed. The design is also well suited for extension, since it is based on a small, well-defined central model.
- A comparison of different data structure types for time constraint graph representation. The properties of each data structure type have been presented, and a general recommendation has been given, taking into account certain properties of the monitored system.

6.3 Future work

There are several data structure-related issues that were not addressed during the performed tests. The identified follow-ups are listed below.

- Hybrid solutions were not considered. It is possible that a more efficient event monitor design could be created by using different types of data structures to represent different relations.

- The current design uses pointers to represent all relations (association). By using actual objects instead of pointers (aggregation) a better result may be reached in some situations.

Apart from these issues, there are a few concerns with the design as presented in this dissertation. Most of these exist because the focus was on tailoring the implementation for the task of evaluating different data representation strategies. Below is a list of proposed improvements.

- The current method of specifying primitive events requires modification of the monitor code to add instantiations of the event type class. Preferably it should be possible to specify static primitive events in the same way as static composite events, i.e., as an event specification. In certain situations it may also be interesting to specify primitive events dynamically, which is not supported in the presented design.
- In its current form, the parser is called by the event monitor whenever composite event types are to be added. The parser also generates actual objects, which are then added to the event monitor object. In order to increase flexibility and enable the monitor to run on hard targets, the parser should be a separate program, which generates object-constructing source code instead of objects. This code could then be downloaded to and compiled on a hard target.

Acknowledgments

"Always two, there are. No more, no less. A master, and an apprentice."

-- *Yoda*

First of all, I would like to thank my master and supervisor, Jonas Mellin, for his invaluable advice and supportive comments. The same can be said of Ragnar Birgisson and Jörgen Hansson, who have both been of great help.

Further thanks goes to the wonderful people in the systems programming class of '96 at the University of Skövde. Their company has made the long days and nights in front of the screen much more bearable. Of these, Robert Nilsson, Anders Biederbäck, Reine Olsson, Birgitta Lindström, Christoffer Brax and Marcus Thuresson deserve special mention, since they have read and commented on this dissertation.

I am also fortunate enough to have a great, understanding family, who have supported me during this project in spite of not seeing me for almost half a year. I would not have made it this far without their faith and support.

Finally, a thanks to the man whose wonderful visions have been a great distraction and an infinite source of inspiration – George Lucas. Keep the Force flowing.

References

Adrion, W.R., Branstad, M.A., and Cherniavsky, J.C. (1982) Validation, verification and testing of computer software, *ACM Computing Surveys* 14(2), pp. 159-192, June 1982.

Burns, A. and Wellings, A. (1989) *Real-Time Systems and Their Programming Languages*, Addison-Wesley.

Chakravarthy, S. and Mishra, D. (1993) Snoop: An Expressive Event Specification Language For Active Databases, *Technical Report UF-CIS-TR-93-007*, Department of Computer and Information Sciences, University of Florida.

Chakravarthy, S., Krishnaprasad, V., Anwar, E., and Kim, S.K. (1994) Composite Events For Active Databases: Semantics, Contexts and Detection, *20th International Conference on Very Large Databases*, pp. 606-617, Santiago, Chile, September 1994.

Chodrow, S.E., Jahanian, F., and Donner, M. (1991) Run-Time Monitoring of Real-Time Systems, *Proc. of the Real-Time Systems Symposium (RTSS'91)*, pp. 74-83. IEEE Computer Society Press.

Gait, J., (1985) A Debugger for Concurrent Programs, *Software - Practice And Experience*, Issue 15(6), June 1985.

ISO (1996) ISO/IEC 14977:1996 Information technology -- Syntactic metalanguage -- Extended BNF.

Kopetz, H. and Veríssimo, P. (1994) *Real-Time and Dependability Concepts*, Distributed Systems 2nd ed), Chapter 16, pp. 411-446, Addison-Wesley.

Laprie, J.C. (ed.) (1994) *Dependability: Basic Concepts and Terminology*, IFIP WG 10.4 - Dependable Computing and Fault Tolerance.

Liu, G., Mok, A.K., and Konana, P. (1998) A Unified Approach for Specifying Timing Constraints and Composite Events in Active Real-Time Database Systems, *4th Real-Time Technology and Applications Symposium (RTAS '98)*.

Liu, G., Mok, A.K., and Yang, E.J., (1999) Composite Events for Network Event Correlation, *Proc. of IM'99*.

Lucas, G. (1977) *Star Wars Episode IV: A New Hope*, 20th Century Fox.

Mellin, J. (1998) Predictable Event Monitoring, *Licentiate Thesis 737*, Department of Computer and Information Science, Linköpings Universitet.

Mok, A.K., and Liu, G. (1997a) Early Detection of Timing Constraint Violation at Runtime, *Proc. of Real-Time Systems Symposium (RTSS'97)*, IEEE Computer Society Press.

Mok, A.K., and Liu, G. (1997b) Efficient Run-Time Monitoring of Timing Constraints, *Proc. of Real-Time Technology and Applications Symposium (RTAS'97)*.

OSE Support Group (1985), 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.

Patel, R., and Davidsson, B. (1994) *Forskningsmetodikens grunder*, Studentlitteratur.

Schütz, W. (1994) Fundamental Issues in Testing Distributed Real-Time Systems, *Real-Time Systems*, Vol. 7, Issue 2, pp. 129-157.

Schneider, F.B. (1994) *Replication Management using the State-Machine Approach*, Distributed Systems 2nd edn), Chapter 7, pp. 169-197, Addison-Wesley.

Webber, A.B. (1992) A Formal Definition of Unnecessary Computation in Functional Programs, *Technical Report TR92-1260*, Cornell University.

Weiss, M.A. (1999) *Data Structures & Algorithm Analysis in C++, 2nd ed*. Addison Wesley Longman, Inc.

Appendix A - syntax of the specification language

The specification of a composite event type is formally described in extended Backus-Naur form (EBNF) (ISO, 1996) as

eventSpec ::=

event eventname **is** (**satisfaction**|**violation**) **of** ’[’

(basicConstr | ({basicConstr (’&&’ | ’||’)} basicConstr)) ’] ’

basicConstr ::=

occTime ’>=’ occTime

occTime ::=

’@(’ primitiveEventType ’,’ arithmeticExpression ’)’ ’>=’

| ’@(’ primitiveEventType ’,’ occTime ’,’ arithmeticExpression ’)’

arithmeticExpression ::=

[number ’*’] ’i’ [(’+’ | ’-’) number]

| [’-’] number

number ::=

{(’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’)}

Appendix B - source code

The first part of this appendix contains the C++ code used for the event monitor prototype. The second part covers the code used with the programs lex and yacc to create the code for the parser. The third part contains the programs used to generate the event specification files for simulations 1 and 2, and the fourth part lists the event specifications that were used in simulation 3.

Part 1 - C++ code

emonitor.h

```
#ifndef EMONITOR_H
#define EMONITOR_H

#include "Chain.h"
#include "edge.h"
#include "vertex.h"
#include "etype.h"

class Timer;
class Cgraph;

// This structure connects an event type with a constraint graph.
// By keeping a  chain of these, the monitor can determine which
// graphs should be notified when an event is signalled to the
// monitor.
typedef struct
{
    char* ename;
    Cgraph* graph;
} eventToGraph;

class Emonitor
{
public:
    Emonitor();
    ~Emonitor();
    void addGraph(Cgraph* cg);
    void removeGraph(Cgraph* cg);
    void addEvent(Etype* ep) {monitoredEvents.add(ep);};
    int getnTimers() {return timerlist.numValues();};
    int getnGraphs() {return monitoredGraphs.numValues();};

    void compileAllGraphs();

    void addEventsFromFile(char* fileName);

    void addConnection(eventToGraph* egstruct);
```

```
    void instantiate(char* ename, int index, int occTime);

    void showAllGraphs();
    void showConnections();

    void addToQueue(Cgraph* tg) {graphQueue.add(tg);};
    void removeFromQueue(Cgraph* tg) {graphQueue.remove(tg);};
private:
    Chain<eventToGraph*> connections;
    Chain<Timer*> timerlist;
    Chain<Cgraph*> monitoredGraphs;
    Chain<Etype*> monitoredEvents;
    Chain<Cgraph*> graphQueue;
};

#endif
```

emonitor.cc

```
#include "ChainItr.h"
#include "emonitor.h"
#include "cgraph.h"
#include <string.h>
#include <stdio.h>

extern yyparse();
extern FILE* yyin;

#define VERBOSE 0

Emonitor::Emonitor()
{
}

Emonitor::~Emonitor()
{
    ChainIterator<Cgraph*> cgitr(monitoredGraphs);
    ChainIterator<Timer*>  titr(timerlist);
    ChainIterator<eventToGraph*> citr(connections);

    for(cgitr.rewind();!cgitr.pastEnd();cgitr++)
        delete (Cgraph*) cgitr.getValue();

    for(titr.rewind();!titr.pastEnd();titr++)
        delete (Timer*) titr.getValue();

    for(citr.rewind();!citr.pastEnd();citr++)
        delete (citr.getValue())->ename;
}

void Emonitor::addConnection(eventToGraph* eg)
{
    BOOL exists=false;
    ChainIterator<eventToGraph*> itr(connections);
    eventToGraph* e2gstruct=0;

    for(itr.rewind();(!itr.pastEnd())&&(!exists);itr++)
    {
        e2gstruct=itr.getValue();
        if((!strcmp(e2gstruct->ename,eg->ename))&&(e2gstruct->graph==eg->graph))
            exists=true;
    }
    if(!exists)
    {
        if(!connections.add(eg))
            cerr<<"Could not add connection"<<endl;
    }
```

```
}

void Emonitor::removeGraph(Cgraph* cg)
{
    ChainIterator<eventToGraph*> itr(connections);
    eventToGraph* e2gstruct=0;

    for(itr.rewind();!itr.pastEnd();itr++)
    {
        e2gstruct=itr.getValue();
        if(e2gstruct->graph==cg)
            connections.remove(e2gstruct);
    }

    monitoredGraphs.remove(cg);
}

void Emonitor::addEventsFromFile(char* fileName)
{
    yyin=fopen(fileName,"r");

    if(VERBOSE) cout<<"Adding events from file "<<fileName<<"..."<<endl;
    yyparse();

    if(VERBOSE) cout<<"Parsing done."<<endl;
}

void Emonitor::instantiate(char* enm, int index, int occTime)
{
    ChainIterator<Cgraph*> gitr(graphQueue);
    ChainIterator<eventToGraph*> itr(connections);
    eventToGraph* cg;

    ChainIterator<Etype*> eitr(monitoredEvents);
    Etype* etp;

    if(monitoredEvents.numValues())
    {
        for(eitr.rewind();!eitr.pastEnd();eitr++)
        {
            etp=eitr.getValue();
            if(!strcmp(etp->getName(),enm))
                etp->instantiate(occTime,index);
        }
    }

    if(connections.numValues())
    {
        for(itr.rewind();!itr.pastEnd();itr++)
        {
            cg=itr.getValue();
```

```
            if(!strcmp(cg->ename,enm))
                (cg->graph)->instantiate(enm, index, occTime);
        }
    }

    // the newly constructed clones are not added to the monitor until
    // all graphs have been instantiated

    if(graphQueue.numValues())
    {
        for(gitr.rewind();!gitr.pastEnd();gitr++)
            addGraph(gitr.getValue());
        graphQueue.clear();
    }
}

void Emonitor::addGraph(Cgraph* cg)
{
    eventToGraph* e2gstruct=0;
    Vertex* vp=0; CEtype* cep;

    if(!monitoredGraphs.add(cg))
        cerr<<"Emonitor::addGraph() : Could not add graph!"<<endl;

    //add connections betweens all events in the graph and the graph
    //so that the monitor quickly can find the graphs that have a
    //vertex associated with a given event

    for(int i=0;i<(cg->getnVertices());i++)
    {
        vp=cg->getVertex(i);
        e2gstruct=new eventToGraph;
        e2gstruct->ename=new char[strlen(vp->getEventID())+1];
        strcpy(e2gstruct->ename,vp->getEventID());
        e2gstruct->graph=cg;
        addConnection(e2gstruct);
    }
}

void Emonitor::showConnections()
{
    ChainIterator<eventToGraph*> itr(connections);
    eventToGraph* e2gstruct;

    cout<<"Connections:"<<endl;
    cout<<"------------"<<endl;
    for(itr.rewind();!itr.pastEnd();itr++)
    {
        e2gstruct=itr.getValue();
        cout<<e2gstruct->ename<<"-->"<<((e2gstruct->graph)
            ->getCEtype())>getName()<<endl;
```

```
    }
    cout<<endl;
}

void Emonitor::compileAllGraphs()
{
    ChainIterator<Cgraph*> itr(monitoredGraphs);

    for(itr.rewind();!itr.pastEnd();itr++)
        itr.getValue()->compile();
}

void Emonitor::showAllGraphs()
{
    int i=0;
    ChainIterator<Cgraph*> cgitr(monitoredGraphs);

    for(cgitr.rewind();!cgitr.pastEnd();cgitr++)
    {
        cout<<"Graph #"<<++i<<":"<<endl;
        cgitr.getValue()->compile();
        cgitr.getValue()->print();
        cout<<endl;
    }
}
```

cgraph.h

```
#ifndef CGRAPH_H
#define CGRAPH_H

#include "vertex.h"
#include "edge.h"
#include "Chain.h"
#include "ChainItr.h"
#include "cetype.h"
#include <iostream.h>

class Edge;
class CEtype;

enum STATE {undefined,satisfied,violated};

class Cgraph
{
friend ostream& operator<<(ostream& os,Cgraph cg);
public:
    Cgraph();
    Cgraph(Cgraph& templ);
    ~Cgraph();

    Vertex* addVertex(Vertex* iv);
    // Returns a pointer to the added vertex if there was
    // no equivalent vertex already in the graph. If there
    // was an equivalent vertex, a pointer to that vertex
    // is returned.

    void addEdge(Edge* ie);
    void addEdge(int vindex1, int vindex2, int weight);
    void print();
    int getEdgeWeight(int vindex1, int vindex2);
    int getDistance(int i, int j);
    BOOL infinite(int i, int j) {return isInfinite[i][j];};
    Vertex* getVertex(int index);
    int getnVertices() {return vertices.numValues();};

    void setTemporary(BOOL arg) {temp=arg;};
    BOOL isTemporary() {return temp;};

    BOOL checkConstraint(int i, int j);
    BOOL checkConstraint(Vertex* from, Vertex* to);

    STATE getState() {return state;};
    BOOL isViolated() {return ((state==violated) ? true : false);};
    BOOL isSatisfied() {return ((state==satisfied) ? true : false);};
```

```
    void setState(STATE istate) {state=istate;};
    void setVariableValue(int value);

    void instantiate(char* ename, int index, int occTime);
    void evaluate();

    void setCEtype(CEtype* itype) {compevent=itype;};
    CEtype* getCEtype() {return compevent;};

    void compile();

private:
    Chain<Vertex*> vertices;
    Chain<Edge*> edges;

    STATE state;

    BOOL temp;

    BOOL** isInfinite;
    int** dist;

    CEtype* compevent;
};
```

cgraph.cc

```
#include "cgraph.h"
#include "ChainItr.h"
#include <memory.h>
#include "vertex.h"
#include <string.h>
#include "emonitor.h"

#define MIN(x,y) (((x)<(y))? (x):(y))

#define VERBOSE 0

class Vertex;

extern Emonitor mainMonitor;

Cgraph::Cgraph()
{
    vertices.clear(); edges.clear(); state=undefined; isInfinite=0; dist=0; temp=false;
}

Cgraph::Cgraph(Cgraph& templ)
{
    int nvertices=(templ.vertices).numValues();
    ChainIterator<Vertex*> vitr(templ.vertices);
    for(vitr.rewind();!vitr.pastEnd();vitr++)
        addVertex(new Vertex(*vitr.getValue()));
    //Note: this constructor does not copy edges!

    state=templ.state;

    isInfinite=new BOOL*[nvertices];
    dist=new int*[nvertices];

    for(int i=0;i<nvertices;i++)
    {
        isInfinite[i]=new BOOL[nvertices];
        dist[i]=new int[nvertices];
        memcpy(isInfinite[i],templ.isInfinite[i],sizeof(BOOL)*nvertices);
        memcpy(dist[i],templ.dist[i],sizeof(int)*nvertices);
    }

    compevent=templ.compevent;
    temp=false;
}

Cgraph::~Cgraph()
{
    if(isInfinite)
    {
```

```
        for(int i=0;i<getnVertices();i++)
            delete isInfinite[i];
        delete [] isInfinite;
    }
    if(dist)
    {
        for(int i=0;i<getnVertices();i++)
            delete dist[i];
        delete dist;
    }


    ChainIterator<Vertex*> itr(vertices);

    for(itr.forward();!itr.pastBeg();itr--)
        delete itr.getValue();

    //edges are deallocated automatically when the vertices
    //are deleted

}

void Cgraph::compile()
{
    int n=getnVertices();

    if(isInfinite)
    {
        for(int i=0;i<n;i++)
            delete isInfinite[i];
        delete isInfinite;
    }
    if(dist)
    {
        for(int i=0;i<n;i++)
            delete dist[i];
        delete dist;
    }

    isInfinite=new BOOL*[n];
    for(int i=0;i<n;i++)
        isInfinite[i]=new BOOL[n];

    dist=new int*[n];
    for(i=0;i<n;i++)
        dist[i]=new int[n];

    for(i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        if(!(dist[i][j]=getEdgeWeight(i,j)))
```

```
            isInfinite[i][j]=true;
        else
            isInfinite[i][j]=false;
    }

    int vsize=vertices.numValues();

    for(i=0;i<vsize;i++)
    {
        dist[i][i]=0;
        isInfinite[i][i]=false;
    }
    for(int k=0;k<vsize;k++)
    {
        if(dist[k][k]<0)
        {
            if(VERBOSE) cout<<"Warning: negative cycle found in event "<<compevent
                ->getName()<<endl;
            return;
        }
        else
        {
            for(i=0;i<vsize;i++)
            {
                for(int j=0;j<vsize;j++)
                {
                    if(isInfinite[i][j]&&(!(isInfinite[i][k]||isInfinite[k][j])))
                    {
                        dist[i][j]=dist[i][k]+dist[k][j];
                        isInfinite[i][j]=false;
                    }
                    else if((isInfinite[i][k]||isInfinite[k][j])&&!isInfinite[i][j])
                    {
                        dist[i][j]=dist[i][j];
                        isInfinite[i][j]=false;
                    }
                    else if((!isInfinite[i][j])&&(!isInfinite[i][k])&&(!isInfinite[k][j]))
                    {
                        dist[i][j]=MIN(dist[i][j],(dist[i][k]+dist[k][j]));
                        isInfinite[i][j]=false;
                    }
                }
            }
        }
    }

    for(i=0;i<vsize;i++)
    {
        if(getVertex(i)->isRel())
        {
            for(int j=0;j<vsize;j++)
```

```
            {
                if((getVertex(j))!=(getVertex(i)->getRef()))
                    isInfinite[i][j]=true;
            }
        }
        else
        {
            for(int j=0;j<vsize;j++)
            {
                if(getVertex(j)->isRel())
                {
                    if((getVertex(i))!=(getVertex(j)->getRef()))
                        isInfinite[i][j]=true;
                }
                else
                {
                    for(int k=0;k<vsize;k++)
                    {
                        if(k!=i&&k!=j&&!isInfinite[i][j]&&!isInfinite[i][k]&&!isInfinite[k][j])
                        {
                            if(dist[i][j]==(dist[i][k]+dist[k][j])
                            && (dist[i][k]<0 || dist[i][k]<=dist[i][j]))
                            {
                                isInfinite[i][j]=true;
                                break;
                            }
                        }
                    }
                }
            }
        }
    }
}

void Cgraph::addEdge(int vindex1, int vindex2, int weight)
{
    addEdge(new Edge(getVertex(vindex1), getVertex(vindex2), weight));
}

int Cgraph::getEdgeWeight(int vindex1, int vindex2)
{
    int retval=0, i=0;
    Vertex* from, *to;
    ChainIterator<Vertex*> citr(vertices);

    if((vindex1<vertices.numValues())&&(vindex2<vertices.numValues()))
    {
        from=getVertex(vindex1);
        to=getVertex(vindex2);

        retval=from->pathLengthTo(to);
```

```
    }

    return retval;
}

int Cgraph::getDistance(int i, int j)
{
    int retval=0;

    if(!dist)
        compile();

    if(isInfinite[i][j])
        cerr<<"Cgraph::getDistance(int,int): distance is infinite!"<<endl;
    else if(i>getnVertices()-1||j>getnVertices()-1||i<0||j<0)
        cerr<<"Cgraph::getDistance(int,int): inaccurate vertice indices"<<endl;
    else
        retval=dist[i][j];

    return retval;
}

BOOL Cgraph::checkConstraint(int i, int j)
{
    //returns true if a the constraint is violated, false if not

    BOOL retval=false; int x=0;
    Vertex* from, *to;

    ChainIterator<Vertex*> itr(vertices);

    if(!dist)
        compile();

    if(i<0||j<0||i>getnVertices()-1||j>getnVertices()-1)
        cerr<<"Cgraph::Check(int,int): invalid vertex indices"<<endl;
    else if(isInfinite[i][j])
        cerr<<"Cgraph::Check(int,int): no path between vertices"<<endl;
    else
    {
        from=vertices.get(i); to=vertices.get(j);

        if(from->isInstantiated())
        {
            if(to->isInstantiated())
            {
                if((from->getInstTime()+dist[i][j])<to->getInstTime())
                {
                    if(VERBOSE) cout<<"CONSTRAINT VIOLATED"<<endl;
                    retval=true;
                    state=violated;
```

```
                }
            }
            else if(dist[i][j]>=0)
            {
                if(VERBOSE) cout<<"Deadline Timer needed for event "<<compevent
                    ->getName()<<endl;
            }
            else
            {
                if(VERBOSE) cout<<"CONSTRAINT VIOLATED"<<endl;
                retval=true;
                state=violated;
            }
        }
        else if(to->isInstantiated() && dist[i][j]<0)
        {
            if(VERBOSE) cout<<"Timeout Timer needed for event "<<compevent
                ->getName()<<endl;
        }
    }
    return retval;
}

BOOL Cgraph::checkConstraint(Vertex* from, Vertex* to)
{
    //returns true if the constraint is violated, false if not

    BOOL retval=false;
    int i=0,j=0;

    if(!dist)
        compile();

    if(!(i=vertices.search(from)))
        cerr<<"Cgraph::Check(Vertex*,Vertex*): first vertex not found"<<endl;
    else if(!(j=vertices.search(to)))
        cerr<<"Cgraph::Check(Vertex*,Vertex*): second vertex not found"<<endl;
    else if(isInfinite[i][j])
        cerr<<"Cgraph::Check(Vertex*,Vertex*): no path between vertices"<<endl;
    else
    {
        if(from->isInstantiated())
        {
            if(to->isInstantiated())
            {
                if((from->getInstTime()+dist[i][j])<to->getInstTime())
                {
                    if(VERBOSE) cout<<"CONSTRAINT VIOLATED"<<endl;
                    retval=true;
                    state=violated;
                }
```

```
            }
            else if(dist[i][j]>=0)
            {
                if(VERBOSE) cout<<"Deadline Timer Needed!"<<endl;
            }
            else
            {
                if(VERBOSE) cout<<"CONSTRAINT VIOLATED"<<endl;
                retval=true;
                state=violated;
            }
        }
        else if(to->isInstantiated() && dist[i][j]<0)
        {
            if(VERBOSE) cout<<"Timeout Timer Needed!"<<endl;
        }
    }
    return retval;
}

void Cgraph::instantiate(char* ename, int index, int occTime)
{
    ChainIterator<Vertex*> itr(vertices);
    Vertex* vp; int i=0; int val;

    for(itr.rewind();!itr.pastEnd();itr++)
    {
        vp=itr.getValue();
        if(!strcmp(vp->getEventID(),ename))
        {
            if(index==vp->getTerm())
            {
                if(vp->getVariable()==' ')
                {
                    vp->instantiate(occTime);
                    if( (i=vertices.search(vp))!=-1)
                    {
                        for(int j=0;j<vertices.numValues();j++)
                        {
                            if(i!=j)
                            {
                                if(!isInfinite[i][j])
                                {
                                    checkConstraint(i,j);
                                }
                                if(!isInfinite[j][i])
                                {
                                    checkConstraint(j,i);
                                }
                            }
                        }
```

```
                        evaluate();
                    }
                }
            }
            else if(vp->getVariable()!=' ')
            {
                // try to find an integer value for i that solves the equation
                // a*i+b=x, where
                // a=vp->getMultiplicator()
                // b=vp->getTerm();
                // x=index
                int rest; int val; Cgraph* tgraph;
                rest=(index-(vp->getTerm()))%(vp->getMultiplicator());
                if(rest==0)
                {
                    val=(index-(vp->getTerm()))/(vp->getMultiplicator());
                    mainMonitor.addToQueue(tgraph=new Cgraph(*this));
                    //mainMonitor.addGraph(tgraph=new Cgraph(*this)); //create clone!
                    tgraph->setTemporary(true);
                    tgraph->setVariableValue(val);
                    ((Cgraph*) tgraph)->instantiate(ename,index,occTime);
                }
            }

        }
    }
}

void Cgraph::setVariableValue(int value)
{
    //This function sets the variable value of all vertices in the graph (note that only
    //one variable is allowed per graph) to the argument value. For example, if this function
    //is called with the value 9, a vertex which represents @(E1,3*i+4) would be
    //changed to represent @(E1,31).

    ChainIterator<Vertex*> itr(vertices);

    for(itr.rewind();!itr.pastEnd();itr++)
        itr.getValue()->setVariableValue(value);
}

void Cgraph::evaluate()
{
    ChainIterator<Vertex*> itr(vertices);

    BOOL status=false;

    if((!isSatisfied())&&(!isViolated()))
    {
        for(itr.rewind();(!itr.pastEnd())&&(!status);itr++)
        {
```

```
            if(!((itr.getValue())->isInstantiated()))
                status=true;
        }
        if(!status)
            setState(satisfied);
    }

    if(isSatisfied()&&(compevent->getCondition()==satisfaction))
    {
        compevent->trigger();  //This is not consistent with the implementation model.
                                //When multiple graphs for a composite event are supported,
                                //this should be replaced with compevent->notify(), or
                                //something similar.
    }
    else if(isViolated()&&(compevent->getCondition()==violation))
        compevent->trigger();  //see above

    if(isTemporary() && (isSatisfied()||isViolated()))
    {
        mainMonitor.removeGraph(this);
        mainMonitor.removeFromQueue(this);
        delete this;
    }
}

Vertex* Cgraph::addVertex(Vertex* iv)
{
    ChainIterator<Vertex*> itr(vertices);
    BOOL exists=false;
    Vertex* tmpvx;

    // Check if an equivalent vertex already exists in the graph.
    // If that is the case, the vertex is not added and a pointer
    // to the equivalent vertex is returned.
    for(itr.rewind();!itr.pastEnd()&&!exists;itr++)
    {
        tmpvx=itr.getValue();
        if((!strcmp(tmpvx->getEventID(),iv->getEventID()))
            &&tmpvx->getVariable()==iv->getVariable()
            &&tmpvx->getMultiplicator()==iv->getMultiplicator()
            &&tmpvx->getTerm()==iv->getTerm())
        {
            exists=true;
            delete iv;
            iv=tmpvx;
        }
    }

    // Otherwise, the vertex is added to the graph.
    if(!exists)
    {
```

```
            vertices.add(iv);
            iv->setGraph(this);
        }

        return iv;
}

void Cgraph::addEdge(Edge* ie)
{
    // Checks if there are any equivalent edges before adding the argument edge.
    // If there are, the argument edge is added if its weight is lower than the
    // existing edge. In that case, the existing edge is removed.

    ChainIterator<Edge*> itr(edges);
    BOOL exists=false;
    Edge* tmpedge;

    for(itr.rewind();!itr.pastEnd()&&!exists;itr++)
    {
        tmpedge=itr.getValue();


        if((ie->getFromVertex()==tmpedge->getFromVertex())&&(ie
            ->getToVertex()==tmpedge->getToVertex()))
        {
            exists=true;
            if(ie->getWeight()<tmpedge->getWeight())
            {
                edges.remove(tmpedge);
                delete tmpedge;
                edges.add(ie);
            }
            else
                delete ie;
        }
    }

    if(!exists)
        edges.add(ie);
}

void Cgraph::print()
{
    int i=0;

    ChainIterator<Vertex*> vitr(vertices);
    ChainIterator<Edge*> eitr(edges);

    cout<<"Displaying graph information:"<<endl;
    cout<<"-----------------------------"<<endl;
    cout<<"Event name: "<<getCEtype()->getName()<<endl;
```

```
    cout<<"Number of vertices: "<<vertices.numValues()<<endl;
    cout<<"Number of edges   : "<<edges.numValues()<<endl<<endl;
    cout<<"------------"<<endl;
    cout<<"((Vertices))"<<endl;
    cout<<"------------"<<endl;
    for(vitr.rewind();!vitr.pastEnd();vitr++)
    {
        cout<<i++<<": ";
        (*vitr.getValue()).print();
        cout<<endl;
    }
    cout<<endl;
    cout<<"---------"<<endl;
    cout<<"((Edges))"<<endl;
    cout<<"---------"<<endl;
    for(eitr.rewind(),i=0;!eitr.pastEnd();eitr++)
    {
        cout<<i++<<": ";
        (eitr.getValue())->print();
    }

    cout<<endl;

    if(dist)
    {
        cout<<"-------------"<<endl;
        cout<<"((distances))"<<endl;
        cout<<"-------------"<<endl;
        for(i=0;i<getnVertices();i++)
        {
            for(int j=0;j<getnVertices();j++)
            {
                cout<<i<<"-->"<<j<<": ";
                if(!isInfinite[i][j])
                    cout<<dist[i][j];
                else
                    cout<<"infinite!";
                cout<<endl;
            }
        }
        cout<<endl;
    }
}

Vertex* Cgraph::getVertex(int index)
{
    Vertex* retval=0;
    int i=0;
    ChainIterator<Vertex*> itr(vertices);

    if(index<vertices.numValues())
```

```
    {
        for(itr.rewind();i!=index;itr++,i++)
            ;
        retval=itr.getValue();
    }

    return retval;
}
```

etype.h

```
#ifndef ETYPE_H
#define ETYPE_H

#include "Chain.h"
#include "ChainItr.h"

class Etype
{
public:
    Etype(); //should not be called by user
    Etype(char* iname);
    ~Etype();
    char* getName() {return name;};
    void instantiate(int time, int index);
    int getHistoryLength();
    int getoccTime(int index);
    void setMaxHistory(int imax) {maxHistory=imax;};
protected:
    int maxHistory;
    Chain<int> history;
    char* name;
    int currIndex;
};

#endif
```

etype.cc

```
#include "ChainItr.h"
#include "Chain.h"
#include "etype.h"
#include "emonitor.h"
#include <string.h>

#define HISTORYDEFAULT 10;
#define VERBOSE 0

class Emonitor;
extern Emonitor mainMonitor;

Etype::Etype()
{
    name=0; currIndex=0;
    history.clear();
    mainMonitor.addEvent(this);
    maxHistory=HISTORYDEFAULT;
}

Etype::Etype(char* iname)
{
    currIndex=0;
    name=new char[strlen(iname)+1];
    strcpy(name,iname);
    history.clear();
    mainMonitor.addEvent(this);
    maxHistory=HISTORYDEFAULT;
}

Etype::~Etype()
{
    delete name;
}

int Etype::getHistoryLength()
{
    return history.numValues();
}

void Etype::instantiate(int time, int index)
{
    if(VERBOSE) cout<<"Instantiating event "<<name<<endl;

    if((history.numValues()>0) && (history.numValues()>=maxHistory))
        history.remove(history.get(0));

    history.add(time);
```

```
    currIndex=index;
}

int Etype::getoccTime(int index)
{
    int retval=0;
    ChainIterator<int> itr(history);

    //index=0 returns the most recent instance
    //index=-1 returns the second most recent instance
    //...

    //positive values of index are not allowed

    if(index>0)
    {
        retval=0;
        cerr<<"Positive index values not allowed when accessing event histories!"<<endl;
    }
    else if(history.numValues()==0)
    {
        retval=0;
        cerr<<"Trying to access non-existent event history of event type: "<<name<<endl;
    }
    else if((-index)>=history.numValues())
    {
        retval=0;
        cerr<<"Trying to access non-existent history entry of event type: "<<name<<endl;
    }
    else if(index==0)
    {
        itr.forward();
        retval=itr.getValue();
    }
    else if(index<0)
    {
        for(itr.forward();index<0;itr--,index++);
        retval=itr.getValue();
    }

    return retval;
}
```

cetype.h

```
#ifndef CETYPE_H
#define CETYPE_H

#include "etype.h"
#include "cgraph.h"

enum CONDITION {satisfaction,violation};

class CEtype : public Etype
{
public:
    CEtype(CONDITION icond);
    CEtype(char* iname, CONDITION icond);
    ~CEtype();
    void setCondition(CONDITION icond) {cond=icond;};
    CONDITION getCondition() {return cond;};
    Cgraph* getGraph(int index);
    int numGraphs();
    void addGraph(Cgraph* gp);
    BOOL evaluate();
    void trigger();
private:
    CONDITION cond;
    Chain<Cgraph*> graphs;
};

#endif
```

cetype.cc

```
#include "cetype.h"
#include "emonitor.h"
#include <string.h>

#define VERBOSE 0

class Emonitor;

extern Emonitor mainMonitor;

CEtype::CEtype(CONDITION icond) : Etype()
{
    cond=icond;
    graphs.clear();
}

CEtype::CEtype(char* iname, CONDITION icond) : Etype(iname)
{
    if(VERBOSE) cout<<"Creating new event "<<getName()<<"!"<<endl;
    cond=icond;
    graphs.clear();
}

CEtype::~CEtype()
{
    Etype::~Etype();
}

Cgraph* CEtype::getGraph(int index)
{
    int i=0;

    ChainIterator<Cgraph*> itr(graphs);

    for(itr.rewind();(!itr.pastEnd())&&(i!=index);itr++,i++)
        ;
    if(!itr.pastEnd())
        return itr.getValue();
    else
        return 0;
}

int CEtype::numGraphs()
{
    return graphs.numValues();
}

void CEtype::addGraph(Cgraph* gp)
{
```

```
    graphs.add(gp);
}

BOOL CEtype::evaluate()
{
    //returns true if the composite event is satisfied, false if not
    //if the event is satisfied, an event occurrence of this event
    //is triggered

    ChainIterator<Cgraph*> itr(graphs);

    BOOL retval=false;

    if(cond==satisfaction)
    {
        for(itr.rewind();!itr.pastEnd()&&!retval;itr++)
            retval=(itr.getValue())->isSatisfied();
    }
    else if(cond==violation)
    {
        for(itr.rewind();!itr.pastEnd()&&!retval;itr++)
            retval=(itr.getValue())->isViolated();
    }

    if(retval)
        trigger();

    return retval;
}

void CEtype::trigger()
{
    //triggers an event occurrence of this event
    mainMonitor.instantiate(name,currIndex+1,1);
}
```

vertex.h

```
#ifndef VERTEX_H
#define VERTEX_H

#include "edge.h"
#include "dtimer.h"
#include "ttimer.h"
#include "nod.h"
#include "cgraph.h"
#include <iostream.h>
#include "Chain.h"

class Edge;
class Timer;
class Cgraph;

class Vertex
{
public:
    Vertex();
    Vertex(Vertexdata vd);
    Vertex(Relvertexdata rvd);
    Vertex(const Vertex& templ);
    ~Vertex();

    void setDeadlineTimer(int time);
    void setTimeoutTimer(int time);
    void addInEdge(Edge* ie);
    void addOutEdge(Edge* oe);

    void merge(Vertex& mv);
    void instantiate(int itime);
    void setVertexData(Vertexdata vd);
    void setVertexData(Relvertexdata rvd);
    virtual void print();

    void setGraph(Cgraph* cg) {parentGraph=cg;};

    char* getEventID() {return eventid;};
    char getVariable() {return variable;};
    int getMultiplicator() {return multiplicator;};
    int getTerm() {return term;};

    void setVariable(char iv) {variable=iv;};
    void setMultiplicator(int im) {multiplicator=im;};
    void setTerm(int it) {term=it;};
    void setVariableValue(int val);

    // Returns 0 if there is no path between *this and
    // the argument vertex. Also returns 0 if this=v.
```

```
    int pathLengthTo(Vertex* v);

    int getnInEdges() {return incomingEdges.numValues();};
    int getnOutEdges() {return outgoingEdges.numValues();};

    BOOL isInstantiated() {return isInst;};

    int getInstTime();

    virtual BOOL isRel() {return false;};
    virtual Vertex* getRef() {return 0;};

protected:
    char* eventid;
    char variable;
    int multiplicator;
    int term;

private:
    Chain<Edge*> incomingEdges;
    Chain<Edge*> outgoingEdges;
    Timer* tim;

    Cgraph* parentGraph;

    BOOL isInst;

    int iTime;
};

#endif VERTEX_H
```

vertex.cc

```
#include "vertex.h"
#include <memory.h>
#include <string.h>
#include "Chain.h"

Vertex::Vertex()
{
    incomingEdges.clear(); outgoingEdges.clear();
    tim=0;
    iTime=-1;   // negative value --> uninstantiated
    eventid=0; variable=' '; multiplicator=0; term=0;
    isInst=false;
}

Vertex::Vertex(Relvertexdata rvd)
{
    incomingEdges.clear(); outgoingEdges.clear();
    tim=0;
    iTime=-1;
    setVertexData(rvd);
    isInst=false;
}

Vertex::Vertex(Vertexdata vd)
{
    incomingEdges.clear(); outgoingEdges.clear();
    tim=0;
    iTime=-1;
    setVertexData(vd);
    isInst=false;
}

Vertex::Vertex(const Vertex& templ)
{
    eventid=new char[strlen(templ.eventid)+1];
    strcpy(eventid,templ.eventid);
    variable=templ.variable;
    multiplicator=templ.multiplicator;
    term=templ.term;

    tim=0;
    isInst=false;
    iTime=0;
}

Vertex::~Vertex()
{
    ChainIterator<Edge*> itr(outgoingEdges);
```

```
    // if every vertex deallocates just its outgoing edges,
    // all edges in the graph will be deallocated when the
    // graph is deleted
    for(itr.rewind();!itr.pastEnd();itr++)
        delete itr.getValue();

    delete tim;
    delete eventid;
}

void Vertex::addInEdge(Edge* ie)
{
    incomingEdges.add(ie);
}

void Vertex::addOutEdge(Edge* oe)
{
    outgoingEdges.add(oe);
}

void Vertex::setVertexData(Relvertexdata rvd)
{
    delete eventid;
    eventid=new char[strlen(rvd.eventid)+1];
    strcpy(eventid,rvd.eventid);
    variable=rvd.variable;
    term=rvd.term;
    multiplicator=rvd.multiplicator;
}

void Vertex::setVertexData(Vertexdata vd)
{
    delete eventid;
    eventid=new char[strlen(vd.eventid)+1];
    strcpy(eventid,vd.eventid);
    variable=vd.variable;
    term=vd.term;
    multiplicator=vd.multiplicator;
}

void Vertex::setVariableValue(int val)
{
    if(variable!=' ')
    {
        term=multiplicator*val+term;
        multiplicator=1;
        variable=' ';
    }
}
```

```
int Vertex::pathLengthTo(Vertex* v)
{
    int retval=0;

    ChainIterator<Edge*> itr(outgoingEdges);

    Edge* tmp;

    for(itr.rewind();!itr.pastEnd();itr++)
    {
        tmp=itr.getValue();
        if(tmp->getToVertex()==v)
            retval=tmp->getWeight();
    }

    return retval;
}

int Vertex::getInstTime()
{
    int retval=0;

    if(isInstantiated())
        retval=iTime;
    else
        cerr<<"Vertex::getInstTime(): Vertex not instantiated!"<<endl;

    return retval;
}

void Vertex::merge(Vertex& mv)
//merges two vertexes
//*this is the new vertex, mv should be deleted
//after calling this function
//(not tested)
{
    ChainIterator<Edge*> itr(mv.incomingEdges);
    ChainIterator<Edge*> jtr(mv.outgoingEdges);

    for(itr.rewind();!itr.pastEnd();itr++)
        addInEdge(itr.getValue());
    for(jtr.rewind();!jtr.pastEnd();jtr++)
        addOutEdge(jtr.getValue());
    //this should perhaps search for duplicate edges
    //and remove the least restricting one
}

void Vertex::setDeadlineTimer(int itime)
{
    tim=new Dtimer(itime, this);
}
```

```
void Vertex::setTimeoutTimer(int itime)
{
    tim=new Ttimer(itime, this);
}

void Vertex::instantiate(int itime)
{
    iTime=itime; isInst=true;
}

void Vertex::print()
{
    cout<<"@("<<eventid<<",";
    if(variable!=' '&&multiplicator!=0)
    {
        if(multiplicator==1)
            cout<<variable;
        else
            cout<<multiplicator<<variable;
    }
    if(term>0&&variable!=' ')
        cout<<"+"<<term;
    else if(term)
        cout<<term;
    cout<<")";
}
```

rvertex.h

```
#ifndef RVERTEX_H
#define RVERTEX_H

#include "vertex.h"
#include <iostream.h>

class Vertex;

class Rvertex : public Vertex
{
public:
    Rvertex();
    ~Rvertex();
    Rvertex(const Vertex& templ);
    void setVertexData(Relvertexdata rvd);
    void setRef(Vertex* vx) {refvertex=vx;};
    Vertex* getRef() {return refvertex;};
    void print();

    virtual BOOL isRel() {return true;};

private:
    Vertex* refvertex;
};

#endif
```

rvertex.cc

```
#include "rvertex.h"

Rvertex::Rvertex() : Vertex()
{
    refvertex=0;
}

Rvertex::Rvertex(const Vertex& templ) : Vertex(templ)
{
}

Rvertex::~Rvertex()
{
}

void Rvertex::setVertexData(Relvertexdata rvd)
{
    Vertex::setVertexData(rvd);
}

void Rvertex::print()
{
    cout<<"@("<<eventid<<",";
    refvertex->print();
    cout<<",";
    if(variable!=' '&&multiplicator!=0)
    {
        if(multiplicator==1)
            cout<<variable;
        else
            cout<<multiplicator<<variable;
    }
    if(term>0&&variable!=' ')
        cout<<"+"<<term;
    else if(term)
        cout<<term;
    cout<<")";
}
```

edge.h

```
#ifndef EDGE_H
#define EDGE_H

#include "vertex.h"
#include <iostream.h>

class Vertex;

class Edge
{
friend ostream& operator<<(ostream& os,Edge& e);
public:
    Edge();          // should not be called by user
    Edge(Vertex* fV, Vertex* tV, int iw);
    Edge(const Edge& templ);
    ~Edge();
    Vertex* getFromVertex() {return fromVertex;};
    Vertex* getToVertex() {return toVertex;};
    int getWeight() {return weight;};
    void setWeight(int nV) {weight=nV;};
    void print();
private:
    Vertex* toVertex;
    Vertex* fromVertex;
    int weight;
};

#endif
```

edge.cc

```
#include "edge.h"

Edge::Edge()
{
    fromVertex=0; toVertex=0; weight=0;
}

Edge::Edge(Vertex* fV, Vertex* tV, int iw)
{
    fromVertex=fV; toVertex=tV; weight=iw;
}

Edge::~Edge()
{}

void Edge::print()
{
    cout<<"From ";
    (getFromVertex())->print();
    cout<<" To ";
    (getToVertex())->print();
    cout<<", ";
    cout<<"Weight: "<<getWeight()<<endl;
}
```

tree.h

```
#ifndef TREE_H
#define TREE_H

#include "nod.h"
#include "cgraph.h"
#include "cetype.h"

class Cgraph;
class CEtype;

class tree
{
public:
    tree();
    tree(nod* topnod);
    ~tree();
    void setTop(nod* topnod);
    nod* getTop();
    nod* getNod(int layer, int index);
    void convertToGraphs(CEtype* cet);
    int get_no_of_layers();
private:
    int no_of_layers;
    nod* top;
};

#endif
```

tree.cc

```
#include "tree.h"
#include "nod.h"
#include <iostream.h>
#include "cgraph.h"
#include "emonitor.h"
#include "Chain.h"
#include "ChainItr.h"

class Cgraph;
class Emonitor;

extern Emonitor mainMonitor;

tree::tree()
{
    top=0;
    no_of_layers=0;
}

tree::tree(nod* topnod)
{
    top=topnod;
    no_of_layers=1+(top->getSubLevels());
}

tree::~tree()
{
    if(top)
        delete top;
}

void tree::setTop(nod* topnod)
{
    top=topnod;
    no_of_layers=1+(top->getSubLevels());
}

nod* tree::getTop()
{
    return top;
}

nod* tree::getNod(int layer, int index)
{
    nod* retnod=0;
    return retnod;
}
```

```
int tree::get_no_of_layers()
{
    return no_of_layers;
}


void tree::convertToGraphs(CEtype* cet)
{
    Cgraph* gr=new Cgraph();

    getTop()->insertInto(gr);

    mainMonitor.addGraph((Cgraph*) gr);
    cet->addGraph((Cgraph*) gr);
    gr->setCEtype(cet);
}
```

nod.h

```
#ifndef NOD_H
#define NOD_H

class Cgraph;

enum NTYPE {left, right, noparent};
enum BOOL {false=0, true=1};

typedef struct
{
    char* eventid;
    char variable;
    int multiplicator;
    int term;
} Vertexdata;

typedef struct
{
    BOOL isRel;
    char* eventid;
    Vertexdata refvertex;
    char variable;
    int multiplicator;
    int term;
} Relvertexdata;

#include "cgraph.h"

class nod
{
public:
    nod();
    nod(char* initval);
    nod(char* initval, nod* v, nod* h);
    nod(Relvertexdata idata1);
    nod(Relvertexdata idata1, Relvertexdata idata2);
    nod(Relvertexdata idata1, Relvertexdata idata2, nod* v, nod* h);
    nod(Relvertexdata idata1, nod* v, nod* h);
    ~nod();
    nod* getParent();
    nod* getVChild();
    nod* getHChild();
    void setVChild(nod* inod);
    void setHChild(nod* inod);
    void connectParent(nod* inod, NTYPE ityp);
    int  getLevel();
    BOOL isTop();
    BOOL isLeaf();
    char* getVal();
```

```
    void setVal(char* newVal);
    void setLevel(int newLevel);
    int  getSubLevels();
    void setSubLevels(int newsub);
    void printRecurs();

    void insertInto(Cgraph* cg);
    // Creates vertices and an edge from the data
    // contained in the node. These are inserted
    // into the argument constraint graph.
    // Then insertInto(cg) is called on all
    // subnodes of this node (if any).

    void setT1data(Relvertexdata idata);
    void setT2data(Relvertexdata idata);
    void setT1data(Vertexdata idata);
    void setT2data(Vertexdata idata);
    Relvertexdata getT1data() {return T1;};
    Relvertexdata getT2data() {return T2;};
    void setD(int nd) {D=nd;};
    int getD() {return D;};
    void setop(char o1, char o2) {op[0]=o1; op[1]=o2;};
    void print();
private:
    int  sublevels;
    char* vaerde;
    nod* vnod;
    nod* hnod;
    nod* parent;
    int level;
    NTYPE typ;

    Relvertexdata T1;
    Relvertexdata T2;
    int D;
    char op[3];
};

#endif
```

nod.cc

```
#include "nod.h"
#include "string.h"
#include <iostream.h>
#include "rvertex.h"

nod::nod()
{
    Relvertexdata r;

    parent=0; vnod=0; hnod=0; vaerde=0; level=0;
    typ=noparent; sublevels=0; D=0; strcpy(op,"  ");

    r.isRel=false;
    r.eventid=0;
    r.variable=' ';
    r.multiplicator=0;
    r.term=0;
    setT1data(r); setT2data(r);
}

nod::nod(char* initval)
{
    Relvertexdata r;

    r.isRel=false;
    r.eventid=0;
    r.variable=' ';
    r.multiplicator=0;
    r.term=0;
    setT1data(r); setT2data(r);

    parent=0; vnod=0; hnod=0; level=0;
    typ=noparent; sublevels=0; D=0; strcpy(op,"  ");

    vaerde=new char[strlen(initval)+1];
    strcpy(vaerde,initval);
}

nod::nod(char* initval, nod* v, nod* h)
{
    Relvertexdata r;

    r.isRel=false;
    r.eventid=0;
    r.variable=' ';
    r.multiplicator=0;
    r.term=0;
    setT1data(r); setT2data(r);
```

```
    parent=0; level=0;
    typ=noparent; D=0; strcpy(op,"  ");
    vaerde=new char[strlen(initval)+1];
    strcpy(vaerde,initval);
    vnod=v;
    hnod=h;
    level=0;
    (v->getSubLevels())>(h->getSubLevels()) ?
        sublevels=1+v->getSubLevels() :
        sublevels=1+h->getSubLevels();
}

nod::nod(Relvertexdata idata1)
{
    Relvertexdata r;
    r.isRel=false; r.eventid=0; r.variable=' ';
    r.multiplicator=0; r.term=0;

    parent=0; vnod=0; hnod=0; vaerde=0; level=0;
    typ=noparent; sublevels=0; D=0; strcpy(op,"  ");
    setT1data(idata1);
    setT2data(r);
}

nod::nod(Relvertexdata idata1, Relvertexdata idata2)
{
    parent=0; vnod=0; hnod=0; vaerde=0; level=0;
    typ=noparent; sublevels=0; D=0; strcpy(op,"  ");

    setT1data(idata1); setT2data(idata2);
}

nod::nod(Relvertexdata idata1, Relvertexdata idata2, nod* v, nod* h)
{
    parent=0; vaerde=0; level=0;
    typ=noparent; D=0; strcpy(op,"  ");

    setT1data(idata1); setT2data(idata2);

    vnod=v; hnod=h;

    (v->getSubLevels())>(h->getSubLevels()) ?
        sublevels=1+v->getSubLevels() :
        sublevels=1+h->getSubLevels();
}

nod::nod(Relvertexdata idata1, nod* v, nod* h)
{
    Relvertexdata r;
    r.isRel=false; r.eventid=0; r.variable=' ';
```

```
    r.multiplicator=0; r.term=0;

    parent=0; vaerde=0; level=0;
    typ=noparent; D=0; strcpy(op,"  ");

    setT1data(idata1); setT2data(r);

    vnod=v; hnod=h;

    (v->getSubLevels())>(h->getSubLevels()) ?
        sublevels=1+v->getSubLevels() :
        sublevels=1+h->getSubLevels();
}

nod::~nod()
{
    if(parent)
        typ==left ? parent->vnod=0 : parent->hnod=0;
    if (vnod)
        delete vnod;
    if (hnod)
        delete hnod;
    delete vaerde;
    if(T1.eventid) delete T1.eventid;
    if(T2.eventid) delete T2.eventid;
    if(T1.isRel && T1.refvertex.eventid) delete T1.refvertex.eventid;
    if(T2.isRel && T2.refvertex.eventid) delete T2.refvertex.eventid;
}

void nod::print()
{
    nod tmpnod;

    if(vaerde)
        cout<<vaerde<<endl;
    else
    {
        cout<<"@("<<T1.eventid<<",";

        if(T1.isRel)
        {
            tmpnod.setT1data(T1.refvertex);
            tmpnod.print();
            cout<<",";
        }

        if(T1.multiplicator==1&&T1.variable!=' ') cout<<T1.variable;
        if(T1.multiplicator>1||T1.multiplicator<0)
        {
            cout<<T1.multiplicator;
            if(T1.variable!=' '&&T1.multiplicator) cout<<T1.variable;
```

```
            if(T1.term>0) cout<<"+"<<T1.term;
            else if(T1.term<0) cout<<T1.term;
        }
        else if(T1.term) cout<<T1.term;
        cout<<")";
        if(D>0) cout<<"+"<<D;
        else if(D<0) cout<<D;
        if(op[0]!=' ') cout<<op[0];
        if(op[1]!=' ') cout<<op[1];
        if(T2.eventid)
        {
            cout<<"@("<<T2.eventid<<",";

            if(T2.isRel)
            {
                tmpnod.setT1data(T2.refvertex);
                tmpnod.print();
                cout<<",";
            }
            if(T2.multiplicator==1&&T2.variable!=' ') cout<<T2.variable;
            if(T2.multiplicator>1||T2.multiplicator<0)
            {
                cout<<T2.multiplicator;
                if(T2.variable!=' '&&T2.multiplicator) cout<<T2.variable;
                if(T2.term>0) cout<<"+"<<T2.term;
                else if(T2.term<0) cout<<T2.term;
            }
            else if(T2.term) cout<<T2.term;
            cout<<")";
        }
    }
}

void nod::insertInto(Cgraph* cg)
{
    Vertex* v1;
    Vertex* v2;
    Edge* e;

    if(T1.eventid)
    {
        if(T1.isRel)
        {
            v1=(Rvertex*) new Rvertex();
            ((Rvertex*) v1)->setRef(cg->addVertex(new Vertex(T1.refvertex)));
        }
        else
            v1=new Vertex();
        v1->setVertexData(T1);
        v1=cg->addVertex(v1);
    }
```

```
    if(T2.eventid)
    {
        if(T2.isRel)
        {
            v2=(Rvertex*) new Rvertex();
            ((Rvertex*) v2)->setRef(cg->addVertex(new Vertex(T2.refvertex)));
        }
        else
            v2=new Vertex();
        v2->setVertexData(T2);
        v2=cg->addVertex(v2);
        e=new Edge(v1,v2,D);
        v1->addOutEdge(e);
        v2->addInEdge(e);
        cg->addEdge(e);
    }

    if(vnod)
    {
        vnod->insertInto(cg);
        hnod->insertInto(cg);
    }
}

void nod::setVChild(nod* inod)
{
    vnod=inod;
}

void nod::setHChild(nod* inod)
{
    hnod=inod;
}

nod* nod::getParent()
{
    return parent;
}

nod* nod::getVChild()
{
    return vnod;
}

nod* nod::getHChild()
{
    return hnod;
}

void nod::setT1data(Vertexdata idata)
{
```

```
    T1.isRel=false;
    if(idata.eventid)
    {
        T1.eventid=new char[strlen(idata.eventid)+1];
        strcpy(T1.eventid,idata.eventid);
    }
    else
        T1.eventid=0;
    T1.eventid=new char[strlen(idata.eventid)+1];
    strcpy(T1.eventid,idata.eventid);
    T1.variable=idata.variable;
    T1.multiplicator=idata.multiplicator;
    T1.term=idata.term;
}

void nod::setT2data(Vertexdata idata)
{
    T2.isRel=false;
    if(idata.eventid)
    {
        T2.eventid=new char[strlen(idata.eventid)+1];
        strcpy(T2.eventid,idata.eventid);
    }
    T2.eventid=0;
    T2.eventid=new char[strlen(idata.eventid)+1];
    strcpy(T2.eventid,idata.eventid);
    T2.variable=idata.variable;
    T2.multiplicator=idata.multiplicator;
    T2.term=idata.term;
}

void nod::setT1data(Relvertexdata idata)
{
    T1.isRel=idata.isRel;
    if(idata.eventid)
    {
        T1.eventid=new char[strlen(idata.eventid)+1];
        strcpy(T1.eventid,idata.eventid);
    }
    else
        T1.eventid=0;
    T1.refvertex=idata.refvertex;
    T1.variable=idata.variable;
    T1.multiplicator=idata.multiplicator;
    T1.term=idata.term;
}

void nod::setT2data(Relvertexdata idata)
{
    T2.isRel=idata.isRel;
    if(idata.eventid)
```

```
    {
        T2.eventid=new char[strlen(idata.eventid)+1];
        strcpy(T2.eventid,idata.eventid);
    }
    else
        T2.eventid=0;
    T2.refvertex=idata.refvertex;
    T2.variable=idata.variable;
    T2.multiplicator=idata.multiplicator;
    T2.term=idata.term;
}

void nod::connectParent(nod* inod, NTYPE ityp)
{
    parent=inod;
    if(ityp==left)
        parent->setVChild(this);
    if(ityp==right)
        parent->setHChild(this);
    typ=ityp;
    if(sublevels+1>parent->getSubLevels())
        parent->setSubLevels(sublevels+1);
}

int nod::getLevel()
{
    return level;
}

BOOL nod::isTop()
{
    return (parent==0 ? true : false);
}

BOOL nod::isLeaf()
{
    return (left==0 ? true : false);
}

char* nod::getVal()
{
    return vaerde;
}

void nod::setVal(char* newVal)
{
    delete vaerde;
    vaerde=new char[strlen(newVal)+1];
    strcpy(vaerde,newVal);
}
```

```
void nod::setLevel (int newLevel)
{
    level=newLevel;
}

int nod::getSubLevels()
{
    return sublevels;
}

void nod::setSubLevels(int newsub)
{
    sublevels=newsub;
    if(parent)
    {
        if(parent->getSubLevels()<1+sublevels)
        parent->setSubLevels(1+sublevels);
    }
}

void nod::printRecurs()
{
    if(vnod)
        vnod->printRecurs();
    if(hnod)
        hnod->printRecurs();
    print();
    cout<<endl;
}
```

Chain.h (dynamic arrays)

```
#ifndef CHAIN_H
#define CHAIN_H

#include <iostream.h>

//Chain.h for Dynamic Arrays

template <class Type>
class Chain {
friend ostream& operator<<(ostream&, const Chain<Type>& aChain);
friend class ChainIterator<Type>;
public:
    Chain();
    Chain(const Chain<Type>& aChain);
    ~Chain();
    int isEmpty();
    void clear();
    Type get(int index) {return arr[index];};
    int search(const Type& aValue);
    int add(const Type& aValue);
    int remove(const Type& aValue);
    int numValues();
private:
    Type* arr;
    int size;
};

#endif
```

Chain.cc (dynamic arrays)

```
#include "Chain.h"
#include "ChainItr.h"
#include <memory.h>

//Chain.cc for Dynamic Arrays

template <class Type>
Chain<Type>::Chain()
{
    arr = 0;
    size = 0;
}

template <class Type>
Chain<Type>::Chain(const Chain<Type>& aChain)
{
    size = aChain.size;
    arr = new Type[size];
    memcpy(arr,aChain.arr,sizeof(Type)*size);
}

template <class Type>
Chain<Type>::~Chain()
{
    delete [] arr;
}

template <class Type>
int Chain<Type>::isEmpty()
{
    return (size==0);
}

template <class Type>
void Chain<Type>::clear()
{
    delete [] arr;
    arr = 0;
    size = 0;
}

template <class Type>
int Chain<Type>::add(const Type& aValue)
{
    int ret = 0;

    if(search(aValue)<0)
    {
```

```
        Type* tmp = new Type[size+1];
        for(int i=0;i<size;i++)
            tmp[i]=arr[i];
        delete [] arr;
        arr = tmp;
        arr[size++] = aValue;
        ret = 1;
    }
    return ret;
}

template <class Type>
int Chain<Type>::remove(const Type& aValue)
{
    int ret = 0;

    if(size > 0)
    {
        int pos = search(aValue);
        if(pos>=0)
        {
            Type* tmp = new Type[size-1];
            for(int i=0;i<pos;i++)
                tmp[i]=arr[i];
            for(;i<size-1;i++)
                tmp[i]=arr[i+1];
            delete [] arr;
            arr = tmp;
            size--;
            ret = 1;
        }
    }
    return ret;
}

template <class Type>
int Chain<Type>::search(const Type& aValue)
{
    int pos = -1;

    for(int i=0;(i<size)&&(pos<0);i++)
    {
        if(arr[i]==aValue)
        pos=i;
    }
    return pos;
}

template <class Type>
int Chain<Type>::numValues()
{
```

```
    return size;
}

template <class Type>
ostream& operator<<(ostream& os, const Chain<Type>& aChain)
{
    os << "Chain output...\n";
    for(int i=0;i<aChain.size;i++)
        os << aChain.arr[i] << endl;
    os << "End Chain output\n";
    return os;
}
```

ChainItr.h (dynamic arrays)

```
#ifndef CHAINITR_H
#define CHAINITR_H

#include "Chain.h"

//ChainItr.h for Dynamic Arrays

template <class Type>
class ChainIterator {
public:
    ChainIterator(Chain<Type>& aChain);
    ChainIterator(ChainIterator<Type>& anIterator);

    Type getValue();
    void rewind();
    void forward();
    int atBeg();
    int atEnd();
    int pastBeg();
    int pastEnd();

    int operator++(int);
    int operator--(int);

    operator Type();

private:
    Chain<Type>& chain;
    int currIndex;
    int pastBegFlag;
    int pastEndFlag;
};

#endif
```

ChainItr.cc (dynamic arrays)

```
#include "ChainItr.h"
#include <stdlib.h>

//ChainItr.cc for Dynamic Arrays

template <class Type>
ChainIterator<Type>::ChainIterator(Chain<Type>& aChain):chain(aChain)
{
    currIndex=0;

    if(aChain.size==0)
    {
        pastBegFlag = 1;
        pastEndFlag = 1;
    }
    else
    {
        pastBegFlag = 0;
        pastEndFlag = 0;
    }
}

template <class Type>
ChainIterator<Type>::ChainIterator(ChainIterator<Type>& anIterator):chain(anIterator.chain)
{
    currIndex = anIterator.currIndex;
    pastBegFlag = anIterator.pastBegFlag;
    pastEndFlag = anIterator.pastEndFlag;
}

template <class Type>
Type ChainIterator<Type>::getValue()
{
    if(chain.size == 0)
    {
        cerr<<"ChainIterator::getValue(): Num values = 0"<<", terminating program!\n";
        exit(1);
    }
    else if(currIndex >= chain.size)
    {
        cerr<<"ChainIterator::getValue(): Num Values = "<<chain.size<<"\nAttempt to access value num ";
        cerr<<currIndex+1<<", terminating program!\n";
        exit(1);
    }
    return chain.arr[currIndex];
}

template <class Type>
```

```
void ChainIterator<Type>::rewind()
{
    currIndex = 0;

    if(chain.size>0)
    {
        pastBegFlag=0;
        pastEndFlag=0;
    }
    else
    {
        pastBegFlag=1;
        pastEndFlag=1;
    }
}

template <class Type>
void ChainIterator<Type>::forward()
{
    currIndex = chain.size-1;

    if(chain.size>0)
    {
        pastBegFlag=0;
        pastEndFlag=0;
    }
    else
    {
        pastBegFlag=1;
        pastEndFlag=1;
    }
}

template <class Type>
int ChainIterator<Type>::atBeg()
{
    return (currIndex==0);
}

template <class Type>
int ChainIterator<Type>::atEnd()
{
    return (currIndex==chain.size-1);
}

template <class Type>
int ChainIterator<Type>::pastBeg()
{
    return pastBegFlag;
}
```

```
template <class Type>
int ChainIterator<Type>::pastEnd()
{
    return pastEndFlag;
}

template <class Type>
int ChainIterator<Type>::operator++(int)
{
    int ret = 0;

    if(currIndex<chain.size-1)
    {
        currIndex++;
        ret = 1;
    }
    else
        pastEndFlag = 1;
    return ret;
}

template <class Type>
int ChainIterator<Type>::operator--(int)
{
    int ret = 0;

    if(currIndex>0)
    {
        currIndex--;
        ret=1;
    }
    else
        pastBegFlag = 1;
    return ret;
}

template <class Type>
ChainIterator<Type>::operator Type ()
{
    return getValue();
}
```

Chain.h (static arrays)

```
#ifndef CHAIN_H
#define CHAIN_H

#include <iostream.h>

//Chain.h for Static Arrays

template <class Type>
class Chain {
friend ostream& operator<<(ostream&, const Chain<Type>& aChain);
friend class ChainIterator<Type>;
public:
    Chain();
    Chain(const Chain<Type>& aChain);
    ~Chain();
    int isEmpty();
    void clear();
    Type get(int index) {return arr[index];};
    int search(const Type& aValue);
    int add(const Type& aValue);
    int remove(const Type& aValue);
    int numValues();
private:
    Type* arr;
    int size;
    int reqsize;
};

#endif
```

Chain.cc (static arrays)

```
#include "Chain.h"
#include "ChainItr.h"
#include <memory.h>

#define ARRSIZE 1965

//Chain.cc for Static Arrays

template <class Type>
Chain<Type>::Chain()
{
    arr = new Type[ARRSIZE];
    size = 0; reqsize=ARRSIZE;
}

template <class Type>
Chain<Type>::Chain(const Chain<Type>& aChain)
{
    size = aChain.size;
    arr = new Type[size];
    reqsize=ARRSIZE;
    memcpy(arr,aChain.arr,sizeof(Type)*size);
}

template <class Type>
Chain<Type>::~Chain()
{
    delete [] arr;
}

template <class Type>
int Chain<Type>::isEmpty()
{
    return (size==0);
}

template <class Type>
void Chain<Type>::clear()
{
    delete [] arr;
    arr = new Type[ARRSIZE];
    size = 0;
}

template <class Type>
int Chain<Type>::add(const Type& aValue)
{
    int ret = 0;
```

```
    if(size>=ARRSIZE)
        cout<<"Chain::add(): Could not add. Required size: "<<reqsize++<<endl;
    else if(search(aValue)<0)
    {
        arr[size++] = aValue;
    ret = 1;
    }
    return ret;
}

template <class Type>
int Chain<Type>::remove(const Type& aValue)
{
    int ret = 0;

    if(size > 0)
    {
        int pos = search(aValue);
        if(pos>=0)
        {
            memcpy(&arr[pos],&arr[pos+1],sizeof(Type)*((--size)-pos));
            ret = 1;
        }
    }
    return ret;
}

template <class Type>
int Chain<Type>::search(const Type& aValue)
{
    int pos = -1;

    for(int i=0;(i<size)&&(pos<0);i++)
    {
        if(arr[i]==aValue)
            pos=i;
    }
    return pos;
}

template <class Type>
int Chain<Type>::numValues()
{
    return size;
}

template <class Type>
ostream& operator<<(ostream& os, const Chain<Type>& aChain)
{
    os << "Chain output...\n";
    for(int i=0;i<aChain.size;i++)
```

```
        os << aChain.arr[i] << endl;
    os << "End Chain output\n";
    return os;
}
```

ChainItr.h (static arrays)

```
#ifndef CHAINITR_H
#define CHAINITR_H

#include "Chain.h"

//ChainItr.h for Static Arrays

template <class Type>
class ChainIterator {
public:
    ChainIterator(Chain<Type>& aChain);
    ChainIterator(ChainIterator<Type>& anIterator);

    Type getValue();
    void rewind();
    void forward();
    int atBeg();
    int atEnd();
    int pastBeg();
    int pastEnd();

    int operator++(int);
    int operator--(int);

    operator Type();

private:
    Chain<Type>& chain;
    int currIndex;
    int pastBegFlag;
    int pastEndFlag;
};

#endif
```

ChainItr.cc (static arrays)

```
#include "ChainItr.h"
#include <stdlib.h>

//ChainItr.cc for Static Arrays

template <class Type>
ChainIterator<Type>::ChainIterator(Chain<Type>& aChain):chain(aChain)
{
    currIndex = 0;

    if(aChain.size == 0)
    {
        pastBegFlag = 1;
        pastEndFlag = 1;
    }
    else
    {
        pastBegFlag = 0;
        pastEndFlag = 0;
    }
}

template <class Type>
ChainIterator<Type>::ChainIterator(ChainIterator<Type>& anIterator):chain(anIterator.chain)
{
    currIndex = anIterator.currIndex;
    pastBegFlag = anIterator.pastBegFlag;
    pastEndFlag = anIterator.pastEndFlag;
}

template <class Type>
Type ChainIterator<Type>::getValue()
{
    if(chain.size == 0)
    {
        cerr<<"ChainIterator::getValue(): Num values = 0"<<", terminating program!\n";
        exit(1);
    }
    else if(currIndex >= chain.size)
    {
        cerr<<"ChainIterator::getValue(): Num Values = "<<chain.size<<"\nAttempt to access value num ";
        cerr<<currIndex+1<<", terminating program!\n";
        exit(1);
    }
    return chain.arr[currIndex];
}

template <class Type>
void ChainIterator<Type>::rewind()
```

```
{
    currIndex = 0;

    if(chain.size>0)
    {
        pastBegFlag=0;
        pastEndFlag=0;
    }
    else
    {
        pastBegFlag=1;
        pastEndFlag=1;
    }
}

template <class Type>
void ChainIterator<Type>::forward()
{
    currIndex = chain.size-1;

    if(chain.size>0)
    {
        pastBegFlag=0;
        pastEndFlag=0;
    }
    else
    {
        pastBegFlag=1;
        pastEndFlag=1;
    }
}

template <class Type>
int ChainIterator<Type>::atBeg()
{
    return (currIndex==0);
}

template <class Type>
int ChainIterator<Type>::atEnd()
{
    return (currIndex==chain.size-1);
}

template <class Type>
int ChainIterator<Type>::pastBeg()
{
    return pastBegFlag;
}

template <class Type>
```

```
int ChainIterator<Type>::pastEnd()
{
    return pastEndFlag;
}

template <class Type>
int ChainIterator<Type>::operator++(int)
{
    int ret = 0;

    if(currIndex<chain.size-1)
    {
        currIndex++;
        ret = 1;
    }
    else
        pastEndFlag = 1;

    return ret;
}

template <class Type>
int ChainIterator<Type>::operator--(int)
{
    int ret = 0;

    if(currIndex>0)
    {
        currIndex--;
        ret = 1;
    }
    else
        pastBegFlag = 1;

    return ret;
}

template <class Type>
ChainIterator<Type>::operator Type ()
{
    return getValue();
}
```

Chain.h (linked lists)

```
#ifndef CHAIN_H
#define CHAIN_H

#include <iostream.h>

//Chain.h for Linked Lists

template <class Type>
class listNode
{
public:
    listNode() {np=0;};
    listNode(Type ival) {value=ival;np=0;};
    void setVal(Type ival) {value=ival;};
    void setNextNode(listNode<Type>* iptr) {np=iptr;};
    Type getVal() {return value;};
    listNode<Type>* getNextNode() {return np;};

    //attaches a node to this node if np=0
    //returns -1 if a node was attached, 0 if not
    int attachNode(Type ival);

    //delete this node and all following nodes in the list
    void deleteRest(); //{if(np) np->deleteRest(); delete this;};
private:
    Type value;
    listNode<Type>* np;
};

template <class Type>
class Chain
{
friend class ChainIterator<Type>;
public:
    Chain();
    Chain(const Chain<Type>& aChain);
    ~Chain() {if(firstNode) firstNode->deleteRest();};
    Type get(int index);
    int numValues() {return size;};
    int isEmpty() {return (size==0 ? 1 : 0);};
    void clear() {if(firstNode) {firstNode->deleteRest(); size=0;}};
    int add(const Type& aValue);
    int remove(const Type& aValue);
    int search(const Type& aValue);
private:
    int size;
    listNode<Type>* firstNode;
};
```

```
#endif
```

Chain.cc (linked lists)

```
#include "Chain.h"
#include <iostream.h>

//Chain.cc for Linked Lists

template <class Type>
listNode<Type>::attachNode(Type ival)
{
    int retval=0;
    if(!np)
    {
        np=new listNode<Type>(ival);
        retval=1;
    }
    return retval;
}

template <class Type>
Chain<Type>::Chain()
{
    size=0; firstNode=0;
}

template <class Type>
Chain<Type>::Chain(const Chain<Type>& aChain)
{
    size=aChain.size;
    listNode<Type>* current=aChain.firstNode;
    listNode<Type>* currentCopy=0;

    if(current)
    {
        firstNode=new listNode(current->getValue());
        currentCopy=firstNode;

        while(current=current.getNextNode())
        {
            currentCopy->np=new listNode(current->getValue());
            currentCopy=currentCopy->getNextNode();
        }
    }
}

template <class Type>
void listNode<Type>::deleteRest()
{
    if(np) np->deleteRest();
```

```
    delete this;
}

template <class Type>
int Chain<Type>::search(const Type& aValue)
{
    int pos=-1; listNode<Type>* lnp=firstNode; int i=1;

    if(size>0)
    {
        if(firstNode->getVal()==aValue)
            pos=0;

        while((lnp=lnp->getNextNode())&&(pos<0))
        {
            if(lnp->getVal()==aValue)
                pos=i;
            else
                i++;
        }
    }

    return pos;
}

template <class Type>
Type Chain<Type>::get(int index)
{
    listNode<Type>* current=firstNode; int i=0;

    if((index>=0)&&(index<size)&&(current!=0))
    {
        while(i<index)
        {
            current=current->getNextNode();
            i++;
        }
        return current->getVal();
    }
    else return (Type) 0;
}

template <class Type>
int Chain<Type>::add(const Type& aValue)
{
    int ret=0; listNode<Type>* lnp=firstNode;

    if(size==0)
    {
        firstNode=new listNode<Type>(aValue);
```

```
        size=1; ret=1;
    }

    else if(search(aValue)<0)
    {
        while(!lnp->attachNode(aValue))
            lnp=lnp->getNextNode();

        ret=1;
        size++;
    }

    return ret;
}

template <class Type>
int Chain<Type>::remove(const Type& aValue)
{
    int ret=0; listNode<Type>* current=firstNode; listNode<Type>* previous;
    int CONTINUE=-1;

    if((size>0))
    {
        if(firstNode->getVal()==aValue)
        {
            //ta bort första noden
            firstNode=firstNode->getNextNode();
            delete current; size--;
        }
        else
        {
            while((current->getNextNode()) && CONTINUE)
            {
                previous=current; current=previous->getNextNode();
                if(current->getVal()==aValue)
                {
                    CONTINUE=0;
                    previous->setNextNode(current->getNextNode());
                    delete current;
                    size--; ret=1;
                }
            }
        }
    }
    return ret;
}
```

ChainItr.h (linked lists)

```
#ifndef CHAINITR_H
#define CHAINITR_H

#include "Chain.h"

//ChainItr.h for Linked Lists

template <class Type>
class ChainIterator
{
public:
    ChainIterator(Chain<Type>& aChain);
    ChainIterator(ChainIterator<Type>& anIterator);

    Type getValue();
    void rewind();
    void forward();
    int atBeg();
    int atEnd();
    int pastBeg();
    int pastEnd();

    int operator++(int);
    int operator--(int);

    operator Type();

private:
    Chain<Type>& chain;
    listNode<Type>* currNode;
    int pastBegFlag;
    int pastEndFlag;
};

#endif
```

ChainItr.cc (linked lists)

```
#include "ChainItr.h"
#include <stdlib.h>

//ChainItr.cc for Linked Lists

template <class Type>
ChainIterator<Type>::ChainIterator(Chain<Type>& aChain):chain(aChain)
{
    currNode=chain.firstNode;

    if(aChain.size == 0)
    {
        pastBegFlag=1;
        pastEndFlag=1;
    }
    else
    {
        pastBegFlag=0;
        pastEndFlag=0;
    }
}

template <class Type>
ChainIterator<Type>::ChainIterator(ChainIterator<Type>& anIterator):chain(anIterator.chain)
{
    currNode=anIterator.currNode;
    pastBegFlag=anIterator.pastBegFlag;
    pastEndFlag=anIterator.pastEndFlag;
}

template <class Type>
Type ChainIterator<Type>::getValue()
{
    if(chain.size==0)
    {
        cerr<<"ChainIterator::getValue(): Num values = 0, terminating program!"<<endl;
        exit(1);
    }
    else if(currNode==0)
    {
        cerr<<"ChainIterator::getValue(): Trying to access nonexistant value"<<endl;
        exit(1);
    }

    return currNode->getVal();
}
```

```
template <class Type>
void ChainIterator<Type>::rewind()
{
    currNode=chain.firstNode;

    if(chain.size>0)
    {
        pastBegFlag=0;
        pastEndFlag=0;
    }
    else
    {
        pastBegFlag=1;
        pastEndFlag=1;
    }
}

template <class Type>
void ChainIterator<Type>::forward()
{
    currNode=chain.firstNode;

    if(currNode)
    {
        while(currNode->getNextNode())
            currNode=currNode->getNextNode();
    }

    if(chain.size>0)
    {
        pastBegFlag=0;
        pastEndFlag=0;
    }
    else
    {
        pastBegFlag=1;
        pastEndFlag=1;
    }
}

template <class Type>
int ChainIterator<Type>::atBeg()
{
    return (currNode==chain.firstNode);
}

template <class Type>
int ChainIterator<Type>::atEnd()
{
    return (currNode->getNextNode()==0);
}
```

```
template <class Type>
int ChainIterator<Type>::pastBeg()
{
    return pastBegFlag;
}

template <class Type>
int ChainIterator<Type>::pastEnd()
{
    return pastEndFlag;
}

template <class Type>
int ChainIterator<Type>::operator++(int)
{
    int ret=0;

    if(!atEnd())
    {
        currNode=currNode->getNextNode();
        ret=1;
    }
    else
        pastEndFlag=1;

    return ret;
}

template <class Type>
ChainIterator<Type>::operator--(int)
{
    int ret=0;
    listNode<Type>* tmpNode=chain.firstNode;

    if(!atBeg())
    {
        while(tmpNode->getNextNode()!=currNode)
            tmpNode=tmpNode->getNextNode();
        currNode=tmpNode;
        ret=1;
    }
    else
        pastBegFlag=1;

    return ret;
}

template <class Type>
ChainIterator<Type>::operator Type()
{
```

```
    return getVal();
}
```

Part 2 - lex and yacc code for the parser

RTL.l (lex code)

```
%{
#include "nod.h"
#include <iostream.h>
#include "y.tab.h"
#include <string.h>

/* Note: primitive event names and keywords must begin with a literal,
composite event names must begin with a capital */
%}

%%

[0-9]+  {yylval.integer=atoi(yytext); return CONSTANT; };

[i-k]    {yylval.character=yytext[0]; return VARIABLE; };

[ \t\n]+ ;

[ \t\n]*[/][/](.*)\n[ \t\n]* ;

[/][*][ \n\t]*(.*)[ \n\t]*[*][/][ \t\n]* ;

[e][vV][eE][nN][tT]                    {return KW_EVENT;};
[i][sS]                          {return KW_IS;};
[o][fF]                          {return KW_OF;};
[s][aA][tT][iI][sS][fF][aA][cC][tT][iI][oO][nN] {return KW_SATISFACTION;};
[v][iI][oO][lL][aA][tT][iI][oO][nN]      {return KW_VIOLATION;};

[A-Z][a-zA-Z0-9]* {yylval.str=new char[strlen(yytext)+1]; strcpy(yylval.str,yytext); return
     STRING;};

\n|.       {return yytext[0];}
```

RTL.y (yacc code)

```
%{
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include "nod.h"
#include "tree.h"
#include "emonitor.h"
#include "cetype.h"
#include "Chain.h"
#include "ChainItr.h"
%};

%union
{
    int integer;
    char character;
    char* str;
    long nodpek;        /* cast to nod* later
                        * nod* is not accepted as type here
                        */
    Relvertexdata rvd;
    Vertexdata vd;
}

%token <integer> CONSTANT
%token <character> VARIABLE
%token KW_EVENT
%token KW_OF
%token KW_SATISFACTION
%token KW_VIOLATION
%token KW_IS
%token <str> STRING/* Must begin with a capital letter */

%type <nodpek> kon
%type <nodpek> dis
%type <nodpek> vertex
%type <nodpek> simplec
%type <nodpek> uttr
%type <rvd> tocc
%type <rvd> rtocc
%type <rvd> occpar
%type <rvd> arithm

%%

cgroup :    constr
        |   constr cgroup
        ;
```

```
constr  :   KW_EVENT STRING KW_IS KW_SATISFACTION KW_OF '[' dis ']'
                    {makeEvent((nod*) $7, $2, (CONDITION) satisfaction);}
        |   KW_EVENT STRING KW_IS KW_VIOLATION KW_OF '[' dis ']'
                    {makeEvent((nod*) $7, $2, (CONDITION) violation);}
        ;

dis :   kon '|' '|' dis         {$$=(long) new nod("|",(nod*) $1,(nod*) $4);}
        |   kon                 {$$=$1;}
        ;

kon:    vertex '&' '&' kon      {$$=(long) new nod("&",(nod*) $1,(nod*) $4);}
        |   vertex              {$$=$1;}
        ;

vertex  :   simplec             {$$=$1;}
        |   '0'                 {$$=(long) new nod("0");}
        ;

simplec     :   uttr '>' '=' tocc       {$$=$1; ((nod*) $$)->setop('>','='); ((nod*) $$)
    ->setT2data($4);}
        ;

uttr:   tocc '+' CONSTANT       {$$=(long) new nod($1); ((nod*) $$)->setD($3);}
        |   tocc '-' CONSTANT       {$$=(long) new nod($1); ((nod*) $$)->setD(-$3);}
        |   tocc                    {$$=(long) new nod($1);}
        ;

tocc    :   '@' '(' STRING ',' occpar ')'  {$$.isRel=false;
                    $$.eventid=new char[strlen($3)+1]; strcpy($$.eventid,$3);
                    $$.refvertex.eventid=0;
                    $$.variable=$5.variable; $$.multiplicator=$5.multiplicator;
                    $$.term=$5.term;}
        |   rtocc           {$$=$1;}
        |   CONSTANT        {$$.isRel=false; $$.variable=' '; $$.multiplicator=0; $$.term=$1;}
        ;

rtocc   :   '@' '(' STRING ',' tocc ',' occpar ')'
                    {$$.isRel=true; $$.eventid=new char[strlen($3)+1];
                    strcpy($$.eventid,$3);
                    $$.refvertex.eventid=new char[strlen($5.eventid)+1];
                    strcpy($$.refvertex.eventid,$5.eventid);
                    $$.refvertex.variable=$5.variable;
                    $$.refvertex.multiplicator=$5.multiplicator;
                    $$.refvertex.term=$5.term;
                    $$.variable=$7.variable; $$.multiplicator=$7.multiplicator;
                    $$.term=$7.term;}
        ;
occpar  :   CONSTANT            {$$.variable=' '; $$.multiplicator=0; $$.term=$1;}
        |   arithm                  {$$=$1;}
        ;
```

```
arithm :    VARIABLE                {$$.variable=$1; $$.multiplicator=1; $$.term=0;}
        |   CONSTANT '*' VARIABLE   {$$.variable=$3; $$.multiplicator=$1; $$.term=0;;}
        |   CONSTANT '*' VARIABLE '+' CONSTANT
                        {$$.variable=$3; $$.multiplicator=$1; $$.term=$5;}
        |   CONSTANT '*' VARIABLE '-' CONSTANT
                        {$$.variable=$3; $$.multiplicator=$1; $$.term=-$5;}
        |   VARIABLE '*' CONSTANT   {$$.variable=$1; $$.multiplicator=$3; $$.term=0;}
        |   VARIABLE '*' CONSTANT '+' CONSTANT
                        {$$.variable=$1; $$.multiplicator=$3; $$.term=$5;}
        |   VARIABLE '*' CONSTANT '-' CONSTANT
                        {$$.variable=$1; $$.multiplicator=$3; $$.term=-$5;}
        |   VARIABLE '+' CONSTANT   {$$.variable=$1; $$.multiplicator=1; $$.term=$3;}
        |   CONSTANT '+' VARIABLE   {$$.variable=$3; $$.multiplicator=1; $$.term=$1;}
        |   VARIABLE '-' CONSTANT   {$$.variable=$1; $$.multiplicator=1; $$.term=-$3;};
        |   '-' CONSTANT      {$$.variable=' '; $$.multiplicator=0; $$.term=-$2;};
        ;

%%

extern Emonitor mainMonitor;

void yyerror(char* s=" ")
{
    fprintf(stderr, "%s\n",s);
}

int yywrap()
{
    return 1;
}

void makeEvent(nod* top, char* name, CONDITION cond)
{
    CEtype* cep;
    /* create a tree from the node information */
    tree* the_tree=new tree(top);

    cep=new CEtype(name, (CONDITION) cond);

    the_tree->convertToGraphs(cep);

    delete the_tree;
}
```

Part 3 - Code for the event specification generation programs

maingenerator.cc (generates main.cc)

```
#include <fstream.h>

#define ABS(x) (((x)<0) ? (-(x)) : (x))

int main()
{
    ofstream fs;
    int nprim=0, ninst=0;

    fs.open("main.cc",ios::in||ios::out);

    cout<<"Number of primitive events    : ";
    cin>>nprim;
    cout<<"Number of event instantiations: ";
    cin>>ninst;

    fs<<"#include <stdio.h>\n#include \"emonitor.h\"\n#include \"etype.h\"\n#include
        <string.h>\n#include <time.h>\n#include \"fstream.h\"\n\n";
    fs<<"extern yyparse();\nextern FILE* yyin;\nEmonitor mainMonitor;\n\n";
    fs<<"int main()\n{\n\tint i;\n\tchar str[10];\n\n\tofstream
         fs;\n\n\tmainMonitor.addEventsFromFile(\"events.spec\");\n";
    fs<<"\tmainMonitor.compileAllGraphs();"<<endl<<endl;
    fs<<"\tfs.open(\"MEASURE/CHAIN/test.txt\",ios::app);\n\n";  /* the result is appended to this file */

    for(int i=0;i<nprim;i++)
        fs<<"\tEtype C"<<i<<"(\"C"<<i<<"\");\n";

    fs<<"\n\tclock();\n\n";

    fs<<"\tfor(i=0;i<"<<ninst<<";i++)\n\t{\n\t\t";
    fs<<"sprintf(str,\"C%d\",i%"<<nprim<<");\n\t\t";
    fs<<"mainMonitor.instantiate(str,(i%"<<nprim<<")+1,i);\n\t}\n\n";
    fs<<"\tfs<<clock()<<endl;\n\n\treturn 0;\n}";

    fs.close();
    return 0;
}
```

specgenerator.cc (generates events.spec)

```
#include <fstream.h>

#define ABS(x) (((x)<0) ? (-(x)) : (x))

int main()
{
    ofstream fs;
    int nevents=0, nconstr=0;

    fs.open("events.spec",ios::in||ios::out);

    cout<<"Number of events to construct      : ";
    cin>>nevents;
    cout<<"Number of constraints in each event: ";
    cin>>nconstr;

    for(int i=0;i<nevents;i++)
    {
        fs<<"event E"<<i<<" is ";
        fs<<((i%2) ? "satisfaction" : "violation")<<" of "<<endl;
        fs<<"["<<endl<<"\t";
        for(int j=0;j<nconstr;j++)
        {
            fs<<"@(C"<<i+j+1<<","<<j+1;
            fs<<")";
            fs<<((j%2) ? '+' : '-')<<(j ? ((i+j)%j)+1 : (i+j))<<">=@(C";
            fs<<((j-i+4) ? ABS(j-i+4) : 1)<<","<<(j==7 ? 1 : ABS(-j+7))<<")";
            fs<<((j==(nconstr-1)) ? "\n" : "&&\n\t");
        }
        fs<<"]"<<endl<<endl;
    }

    fs.close();
    return 0;
}
```

Part 4 - event specifications used for simulation 3

a) (with cloning)

```
event E0 is violation of
[
    // The nodes are identical in order to ensure that the clone is removed immediately. This way,
    // the number of graphs in the system never exceeds 1 when searching the list of graphs.
    @(C1,i)+10>=@(C1,i)
]
```

b) (without cloning)

```
event E0 is violation of
[
    @(C1,1)+10>=@(C1,1)
]
```

Appendix C - interfaces of the central classes

Event monitor

function: addGraph

arguments: constraint graph pointer

returns: void

description: Adds the graph sent as argument to the monitor's list of graphs

function: removeGraph

arguments: constraint graph pointer

returns: void

description: Removes the graph sent as argument to the monitor's list of graphs

function: instantiate

arguments: event type name, event occurrence index, event occurrence time

returns: void

description: Instantiates the event type that matches the event type name sent as argument. This is done by forwarding the index and occurrence time to the relevant event type object. The monitor then searches its list of graphs, and forwards the arguments to any graphs interested in the event by means of the instantiate function in the constraint graph class.

function: compileAllGraphs

arguments: none

returns: void

description: Optimizes all graphs supervised by the monitor. This is typically done after the parsing phase, when the newly constructed graphs have been added to the monitor.

Constraint Graph

function: compile

arguments: none

returns: void

description: Optimizes the graph. This is done by using the graph compilation

algorithm presented by Mok and Liu (1997b) (see section 4.3.6).

function: checkConstraint

arguments: two vertices in the graph that are representing a time constraint

returns: boolean

description: Checks the constraint represented by the relation between the argument vertices using Mok and Liu's constraint check satisfiability algorithm. If the constraint is violated, this function returns true, otherwise it returns false.

function: getState

arguments: none

returns: state of graph ∈ {satisfied,violated,undefined}

description: Returns the current state of the graph. If a violation of a constraint in the graph has been detected during the graph's lifetime, this function returns violated. If all vertices in the graph have been instantiated without any detected violations, this function returns satisfied. In all other cases, it returns undefined.

function: setVariableValue

arguments: *i* value

returns: void

description: Replaces all arithmetic indices (on the form $a*i+b$) in the graph's vertices with the corresponding absolute value for i=<argument value>. For example, if this function is called with an argument value of 9, the arithmetic index $3*i+5$ would be replaced with 32 (3*9+5).

function: instantiate

arguments: event type name, event occurrence index, event occurrence time

returns: void

description: Searches the graph's vertices for any that represent an occurrence function that matches the event type name and index. If such a vertex is found, it is instantiated to the occurrence time sent as argument, and the graph is checked for constraint violations.

function: evaluate

arguments: none

returns: void

description: Determines the current state of the graph (any of {satisfied,violated, undefined}).

Event type

function: instantiate

arguments: event occurrence index, event occurrence time

returns: void

description: Creates an instance of the event type by adding an event occurrence to the event type's history.

Composite event type

function: trigger

arguments: none

returns: void

description: Triggers an event occurrence of this event. This is done by calling the function instantiate in the Event monitor class.

function: evaluate

arguments: none

returns: void

description: Examines the states of the corresponding graphs, and determines if an event occurrence of this event type should be triggered.

Chain

function: add

arguments: a value of the specified data type

returns: 1 if the argument value was successfully added, otherwise 0

description: Adds the argument value to the set if the value does not already exist in the set.

function: remove

arguments: a value of the specified data type

returns: 1 if the argument value was successfully removed, otherwise 0

description: Searches the set for the argument value. If the value is found, it is removed.

function: numValues

arguments: none

returns: the number of values in the set

description: Returns the number of values in the set

function: clear

arguments: none

returns: void

description: Removes all values in the set

function: get

arguments: value index

returns: the value at the position index sent as argument

description: Returns the value at the specified position in the set

ChainIterator

function: rewind

arguments: none

returns: void

description: “Rewinds” the ChainIterator by setting it to point at the first entry in the corresponding Chain.

function: forward

arguments: none

returns: void

description: Sets the ChainIterator to point at the last entry in the corresponding Chain.

function: getValue

arguments: none

returns: the Chain value at the current position

description: Returns the Chain entry at the position currently pointed at by the ChainIterator.

function: pastBeg

arguments: none

returns: true if the pointer has moved past the beginning of the Chain, otherwise false

description: Returns true if the pointer has been moved past the beginning of the corresponding Chain (by means of operator--). Otherwise, it returns false.

function: pastEnd

arguments: none

returns: true if the pointer has moved past the end of the Chain, otherwise false

description: Returns true if the pointer has been moved past the beginning of the corresponding Chain (by means of operator++). Otherwise, it returns false.

function: operator--

arguments: none

returns: void

description: Moves the pointer to the previous entry in the corresponding Chain.

function: operator++

arguments: none

returns: void

description: Moves the pointer to the next entry in the corresponding Chain.

Appendix D - this work and the theory of science

This work has been conducted in a scientific way, and is thus subject to the theory of science. This appendix gives a brief description of the methods and philosophies that we have used in this work.

Patel and Davidsson (1994) describe two major philosophies that can be applied on scientific work -- hermeneutics and positivism.

Hermeneutics is a philosophy that is mainly used in social sciences. A scientist with a hermeneutic viewpoint is not so much interested in actual information as he is in the possible interpretations of information. Hermeneutics supports subjectivism, and the focus is not on proving or falsifying a given theory, but instead on finding the most likely way of interpreting generated data and/or existing information in the research area.

Positivism, on the other hand, advocates a more objective viewpoint, and is most common in the natural sciences. The results of a positivistic work should be hard facts, which should preferably be verifiable by mathematical reasoning. The language used should be clear and non-ambiguos, and should contain as few subjective comments as possible.

Our work is positivistic. The focus of the work is efficiency of an event monitor, which is an observable property of an object in the physical world. This is typical for a positivistic work.

According to Patel and Davidsson (1994) there are several ways to dispose an investigation. These are; survey, case-study, and experiments. By Patel and Davidsson's definitions, this work is experimental in nature, since we study how a variable (the time-efficiency of an event monitor) is affected by different configurations of the event monitor.