Improving Testability of Applications in Active Real-Time Database Environments

HS-IDA-MD-98-001

Ragnar Birgisson (ragnar@ida.his.se)
Institutionen för datavetenskap
Högskolan i Skövde, Box 408
S-54128 Skövde, SWEDEN

Improving Testability of Applications in Active Real-Time Database Environments

Ragnar Karl Birgisson

Submitted by Ragnar Karl Birgisson to the University of Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the Department of Computer Science.

September 1998

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

_____

Ragnar Karl Birgisson

Abstract

The test effort required to achieve full test coverage of an event-triggered real-time system is very high. In this dissertation, we investigate a method for reducing this effort by constraining behavior of the system. We focus on system level testing of applications with respect to timeliness. Our approach is to define a model for constraining real-time system to improve testability. Using this model the applicability of our constraints is easily determined because all the pertinent assumptions are clearly stated. We perform a validation of a test effort upper bound for event-triggered real-time systems with respect to this model. Effects that constraints for improving testability have on predictability, efficiency, and scheduling are investigated and validated. Specific design guidelines for selection of appropriate constraint values are presented in this work. Finally, we discuss mechanisms for handling constraint violations.

Contents

Chapter 1

Introduction

Verification is a fundamental part of a software product's lifecycle. Testing is a dynamic verification technique where the software is executed to see if its behavior complies with its specification. Testing is the most widely used verification technique today. One of the reasons is that testing is applicable to software of any size or complexity which is currently not the case for other methods such as formal verification. Testing is also a well established research area and has been extensively studied within academy and industry [Bei90, Mye79, Het88, OB87, Glas80, Par90, Duk89].

1.1 Testability of Real-Time Systems

A real-time system operates both in the value and time domain. The reason is that operations performed by a real-time systems are often associated with explicit time constraints, e.g., a robot arm must be actuated within milliseconds after an object has passed a sensor. Testing real-time systems is important because these often operate in environments where failure may have catastrophic consequences, e.g., a chemical factory, airplane, or a nuclear plant. Testability is a measurement of how easily an application can be tested. One measurement of testability is the effort required for testing. As the effort involved with testing increases testability decreases. The test effort required to achieve full test coverage, i.e., testing of all possible behaviors, is very large for all but the smallest applications. In a concurrent program the number of synchronization sequences, i.e., the sequences of synchronization events that uniquely determine the results of a concurrent program execution, is enormous for even modest size applications. The first step in improving testability is to bound the number of possible behaviors. Mellin [Mel96] has shown how to improve testability of event-triggered real-time systems by applying constraints on behavior of the system. More

precisely, the following constraints are introduced: i) A bound on the number of times a transaction can be preempted; ii) A bound on the number of concurrently executing transaction instances triggered by the same event type; and iii) Points in time when the environment can be observed are constrained. These constraints result in an upper bound for the test effort of event-triggered real-time system.

1.2 Approach

In this work, Mellin's constraints are investigated in order to determine: i) correctness of the upper bound; ii) effects on predictability, efficiency, and scheduling; iii) feasible constraint value selections; and iv) ways to enforce the constraints in an active real-time database system. The focus in this dissertation is on testing of timeliness, i.e., the system's ability to complete transactions before their deadlines. As a result, system testing, i.e., testing a complete system, is the primary concern. Timeliness of a real-time system cannot be tested until all parts are implemented and integrated into a final system.

1.3 Results

The work has resulted in a model for real-times system that are constrained to improve testability. The model defines properties of transactions, how transaction execution and resource allocation is done, and procedures for testing. A corrected and more accurate upper bound for the test effort of event-triggered real-time systems has been produced. This bound is still high and ways to further reduce the test effort by only completely testing hard transactions are presented. A structured evaluation of effects that the constraints have on predictability, efficiency, and scheduling which shows feasibility of improving testability using these constraints. Ensuring that the system operates within the bounds defined by the constraints requires preventing, avoiding, or detecting and recovering from constraint violations. Each of these method has been evaluated, and solutions suggested. Finally, guidelines for selection of constraint values are presented.

1.4 Overview of Dissertation

In the following chapter an introduction to real-time systems, database systems, and testing is given. Chapter 3 defines the problem at hand, and presents a model for real-time systems that are subjected to constrains in order to improve testability. In chapter 4, existing constraints, for improving the testability of event-triggered real-time systems, are validated and refined. Chapter 5 investigates the effects of these constraints on predictability, efficiency, and scheduling. Guidelines for selecting constraint values are presented, and methods for enforcing the constraints are also discussed. Results of this work are presented in chapter 6. Finally, in chapter 7 conclusions and contributions of this work are described.

Chapter 2

Background

2.1 Real-Time Systems

Applications in which timely response to events in the environment is important appear in various domains. For example, an application that controls robots on a factory floor has to react to events in a predictable and efficient manner to ensure that the movements of the robots does not cause harm. Flight control applications are another example in which correct and timely response is imperative. Several definitions of real-time systems have been suggested [You82, RLK+95, VK93]. The common factor in these is a need to produce results in a timely manner and tight interaction with the physical environment. Stankovic [Sta88], defines a real-time system in the following way:

> "In real-time computing the correctness of the system depends not only on the logical result of a computation, but also on the time at which the results are produced."

This definition highlights that real-time systems not only work in the value domain but also in the time domain. A task is defined by Cheng and Stankovic [CS87] as: "a software module that can be invoked to perform a particular function". Tasks in a real-time system have a deadline which indicates the point in time when the task must be completed. According to Burns [Bur91] and Hanson [Han94], it is common to separate between three types of deadlines: hard, firm, and soft. Missing a hard deadline incurs a penalty much higher than the expected gain of completing the task and the consequences may be catastrophic. A firm deadline can occasionally be missed since the penalty is not

as severe as for hard deadlines. Missing a soft deadline reduces the value of completing the task, but it can still be beneficial to eventually complete this task.

Tasks in real-time systems can be periodic or aperiodic. A periodic task occurs once during its period and its request time, i.e., point in time when a requests is made for execution of the task, is always known a priori. The request time for an aperiodic task is not known a priori and there is no guarantee on the time between two requests. Tasks for which no a priori knowledge about request times is provided, but a minimum time between two requests is guaranteed are termed sporadic. An example of a periodic task is one that monitors the temperature of an engine. At fixed points in time the temperature of the engine is read to ensure that it does not overheat. Aperiodic tasks are associated with dynamic occurrences such as an alarm that is triggered if a leak is detected in a chemical plant.

2.1.1 Design Paradigms for Real-Time Systems

Two ways of designing a real-time system are described by Kopetz [Kop91]: time-triggered and event-triggered. When designing a time-triggered system rigorous assumptions about the environment and its evolution are made. All resource allocation and scheduling in a time-triggered system is done off-line, i.e., before the system is put into operation.

A time-triggered system observes the environment and performs actions at pre-specified points in time. At each observation point the system can perceive which events have occurred since the last observation. A computation in a time-triggered system must finish in the interval between two action points. A time-triggered system can guarantee that all tasks meet their deadlines as long as assumptions about the environment hold. Consequently, most safety-critical systems, i.e., systems where a failure may result in loss of lives or valuable properties are time-triggered.

Event-triggered system react immediately to events in their environment. The set of assumptions about the environment is not as rigid for event-triggered systems as for time-triggered. Kopetz points out that event-triggered systems are more flexible and extensible than time-triggered systems. An event-triggered system can still provide

service if it is pushed beyond the estimated worst-case load, while a time-triggered system cannot. Further, the dynamic nature of event-triggered systems allows them to deal with changes in their environment such as changes to the task set or execution times. In contrast, the static nature of time-triggered systems requires the system to be reconfigured if such changes are needed. According to Stankovic [Sta90], predictability is a central issue in real-time systems and is harder to gurantee in event-triggered systems. As a result it is more difficult to ensure timeliness, i.e., the ability to complete transactions before or at their deadlines, in an event-triggered system. Also, Schütz [Sch91] points out that, due to restrictions applied to time-triggered systems, testing requires less effort. Figures 1.1a and 1.1b illustrate the difference between the time-triggered and event-triggered concepts. As shown in figure 1a, in a time-triggered system a clock interrupt is used to mark the beginning and end of an observation interval. Tasks (T) triggered by events (E) that arrive in interval i are detected and executed in interval i+1. In contrast, event-triggered systems react to events as they occur.

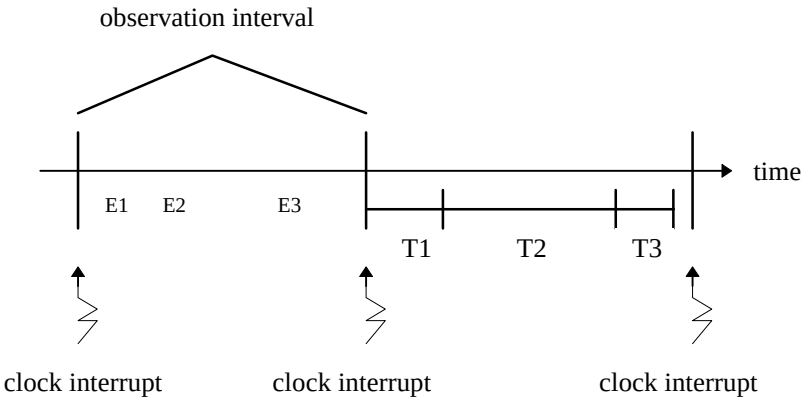


Figure 1.1a. Time-Triggered System

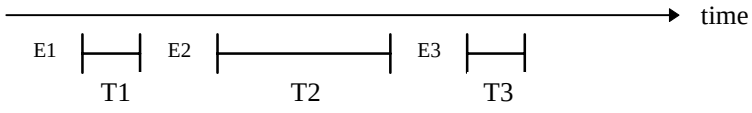


Figure 1.1b. Event-Triggered System

2.1.2 Real-Time Scheduling

One of the most important aspects of real-time systems is scheduling of tasks. A real-time task has an associated deadline which represents the time at which it must be completed. The objective of the scheduler is to execute tasks in such a way that all deadlines are met. There are two main approaches to scheduling: static and dynamic. Further, scheduling can be preemptive or nonpreemptive. A preemptive scheduler can interrupt the currently executing task if another more urgent task arrives. In the nonpreemptive case the executing task cannot be interrupted and holds the processor until it decides to release it. Nonpreemptive scheduling is more difficult than preemptive scheduling, and according to Stankovic and Cheng [SC87], many nonpreemptive scheduling problems have been shown to be NP-hard.

Static scheduling has been thoroughly researched, e.g., Liu and Layland [LL73], Horn [Hor74], Martel [Mar82], and Sha et al [SL86]. According to Kopetz [Kop93], static scheduling is based on rigorous analysis of the system to obtain sufficient a priori knowledge about expected system behavior. Information about resources, precedence constraints, and synchronization requirements of all tasks is used to create a schedule that guarantees all deadlines. During run-time this schedule is executed by a simple dispatcher.

Dynamic scheduling can also make use of off-line analysis, but all scheduling decisions are made on-line, i.e., once the system is running. Dynamic schedulers do not possess the same kind of future knowledge as static scheduling and must react to changes in the system as they occur. Various dynamic scheduling algorithms have been suggested, e.g., rate monotonic [LL73], earliest deadline first, and least laxity first[1]. These select the next task to run based on their priority. For example, the rate-monotonic algorithm schedules tasks according to static priorities which are determined by the periods of tasks. Other algorithms such as earliest deadline first and least laxity first, assign dynamic priorities based on deadlines or laxity of tasks. These algorithms have in common that they only focus on which transaction to execute next. Scheduling

[1] Laxity = deadline - worst-case execution time

algorithms that use additional information to schedule ahead have appeared in recent years, e.g., Ramaritham et. al. [RSS90] and Hansson [Han94].

Most tasks need to share resources of some kind. To ensure integrity of data objects it must be ensured that only one task updates the object at a time. A task that requires access to a data object needs to acquire a lock on the data object before accessing it. This means that a task holding an exclusive lock delays all other tasks that wish to access the same data object. Moreover, due to a phenomenon termed priority inversion, the time a task can delay the execution of another task is not predictable. To explain priority inversion consider the following example given by Sha et. al. [SRL90]. Let T1, T2, T3 be three tasks, where T1 has the highest priority and T3 the lowest. T1 and T3 require access to a common resource. If T3 has acquired exclusive access to the resource when T1 is requests it T1 must wait until T3 releases the resource. During this time T2 may request service. Since T2 has a higher priority than T3 it is granted the processor, and delays the execution of T1 further. The priority ceiling protocol [SRL90] is one way to solve this problem.

2.1.3 Imprecise Computation Techniques

Imprecise computation techniques make it possible to shorten execution of a transaction. This area has been actively researched in recent years, e.g., Liu et. al. [Liu93], Vrbsky et. al. [Vrb91], Liestman and Campbell [LC86]. Three major methods exist for calculating imprecise results. Monotone computations using milestones are applicable when all tasks are monotone. According to Liu et. al. [Liu93], a task is monotone “if the quality of its intermediate result does not decrease as it executes longer”. The method used for monotone tasks is to record intermediate results at appropriate instances in its execution. If the task is prematurely terminated the latest recorded result is used. Another method is to skip certain computation steps in order or to reduce the computation time required. For example, filtering noise in an input results in a higher quality result, but is not necessarily of paramount importance. The third method is to use multiple versions such as in the work by Lin et. al. [LNL87]. In this approach the system has two versions of each task. A primary and an alternate version. The primary version will produce a precise

result, while the alternate version will produce an imprecise result in less time. This method requires that computation and resource requirements of alternate version are bounded.

Imprecise computation techniques have been applied within real-time scheduling to handle transient overload[2], e.g., Hanson [Han94] and Liu et. al. [Liu93]. The multiple version approach is most commonly used. Under normal conditions the primary task is executed, but if the scheduler cannot guarantee the primary the alternate version is used instead.

2.1.4 Flow Control of Events

Controlling flow of events is an important aspect of real-time systems. The behavior of the environment can in many cases not be completely predicted, and situations where more events arrive than the system can handle occur. Flow control mechanisms that regulate flow of events from the environment to the computing element are therefore used. Recently the notion of adaptive filters has been introduced by Stankovic [Sta95]. These are flow control filters that can use information about the state of the system to decide how surplus events are filtered out. Instead of blindly filtering events during overflow these filters can, according to some policy, dynamically decide which events to filter out. Further, polices can change dynamically, and the rate of delivery into the system is based on the impact of meeting deadlines, rather than average throughputs. The main features of these filters are according to Stankovic [Sta95]:

- buffering inputs to temporarily hold unexpected inputs.
- analyzing these inputs to decide what higher level processing is required.
- purging inputs to avoid unnecessary subsequent work and to remove less valuable work when there is overload.

[2] Occurs when execution time of transactions exceeds the available cpu-time.

- having parameters and policies of the filters itself adjustable from higher level policy modules.

According to Stankovic, these filters would primarily be placed between the environment and the real-time control system. But he also notes that filters could be placed internally in the system. The advantage of placing filters inside the system is access to the state of the system allowing them to make more sophisticated decisions. Filters placed in the environment are restricted to performing more primitive operations. Facilities to provide internal filters with knowledge about the state of the system need to be in place. These ideas have been included in the SPRING kernel [SR93] through its reflective architecture [SR95]. Some of the information required for a reflective architecture includes [SR95]:

- importance of a task.
- time requirements.
- time profiles, such as worst-case execution times, deadlines, etc.
- resource needs.

2.2 Real-Time Database Systems

As the amount of information that is used within a computer system increases, the need for efficient organization and manipulation of data grows. According to Ramamritham [Ram93], apart from organizing data and facilitating efficient access, a database system must maintain the integrity and consistency of data as well as support correct transaction execution in the presence of concurrency and failures. Moreover, real-time databases have different requirements than traditional databases. Traditional databases aim at achieving good throughput or response times. For example, maximize the number of executed transactions and minimize average response time for quires. In contrast, the primary goal of a real-time databases system is to meet time constraints, i.e., ensure that all transactions meet their deadlines.

2.2.1 Concurrency Control

If several transactions are allowed to execute concurrently methods to maintain correctness of the database are needed. The reason is that not all interleaving of transactions result in a correct state of the database. Elmasri and Navathe [EN94] give examples of problems that can arise.

- **The lost update problem**. This occurs when two transactions read and write to the same database object and their execution is interleaved in such a way that the database is left in an incorrect state.

- **The temporary update problem**. This problem arises when a transaction T1 updates some data item and is then rolled back for some reason. Before the data item is set back to its original value another transaction reads the modified dirty value (i.e. a value created by a transaction that has not committed).

- **Incorrect summary problem**. This occurs when one transaction is performing a summary function on a set of values while another transaction updates some of these values.

When transaction execute concurrently the order in which operations from various transactions are executed in forms a schedule. A schedule is serializable if it produces the same final state of the database as if each constituent transaction executed serially.

Concurrency control protocols are either pessimistic or optimistic [EN94, GR93]. A pessimistic concurrency protocol guarantees consistency by disallowing non-serializable schedules. This is usually done by locking data items to prevent multiple transactions from accessing the items concurrently. Locks are either shared or exclusive. Several transactions can read an object with a shared lock, but only a single transaction can hold an exclusive lock. The granularity of a lock can range from a single database record to the whole database. Two-phase locking is an example of a pessimistic concurrency

protocol [EN94]. A conservative two-phase locking protocol requires all transactions to lock all items is accesses before execution. This prevents deadlocks from occurring [EN94].

An optimistic concurrency control protocol allows execution of non-serializable schedules. Before committing a transaction the protocol executes validation phase where violation serializability is checked. If serializability is violated the transaction is aborted.

2.2.2 Main Memory Databases

To avoid unpredictability associated with disk access, real-time systems often use main memory based databases. In a main memory database system (MMDB) the entire or partial contents of the database are placed in main memory. As pointed out by Garcia-Molina and Salem [GMS92], an MMDB should not be confused with a disk-resident database with a large cache. MMDB's are specially designed to take full advantage of the data being stored in main memory. One of the impacts of a MMDB is that the lock granule can be increased. The reason for this is that transactions in a MMDB take much less time than in a disk-resident database and, thus, using a larger lock granule will not result in a greater lock contention.

2.2.3 Active Real-Time Databases

Traditional database systems are passive by nature. They execute queries and transactions only when requested to do so. Chakravarthy [Cha89] defines an active database management system as:

"An active DBMS is characterized by its ability to monitor and react to both database events and non-database events in a timely and efficient manner."

An active database system can, for example, automatically execute a consistency check after each update to a certain data object. An example of a non-database event that an active database system can react to is the passage of time.

Active database systems have been thoroughly researched in recent years, e.g., Hanson [Han89], Berndtsson [Ber91], Stonebreaker et. al. [SJG+90], Gatziu et. al. [GGD91]. The active behavior is modeled using event-condition-action (ECA) rules. The semantics of an ECA rule are that when an event occurs the condition is evaluated, and if satisfied the action is executed.

An active database needs to be able to detect and monitor events. For this purpose a special event monitor component is be used. The event-monitor detects the occurrences of events and informs other components, e.g., a rule manager component. A rule manager component is needed to handle condition evaluation and action execution. When the rule manager receives an event instance from the event-monitor it must decide if the associated actions should be taken.

The semantics of ECA rules fit nicely into the event-triggered paradigm and some research into active real-time database systems has been done, e.g., Andler et. al. [AHEM95,AHE+96], Chakravarthy et. al. [CBB+89] and Buchmann et. al. [BBKZ93].

An active real-time database system can be useful for testing purposes. Two important advantages are discussed by Mellin [Mel96]:

- Event Monitor: The active database has a built-in event monitor. This makes it possible to resolve the observability problem (see Sec. 2.4), since the event monitor is an integral part of the system.

- Database storage: Since we have a database at our disposal, storing various data related to testing becomes easier.

Mellin also discusses how transaction support and event criticality [BH95] in active real-time database systems can be used to reduce the test effort. Transaction support,
can in certain situations, generate a looser coupling between time and value. The notion of event criticality, as presented by Berndtsson and Hansson [BH95], means that each event carries information about its criticality which is determined by hardness of actions that this event can trigger. Using event criticality can reduce the required test effort since

only transaction with hard deadlines need to be tested if it is guaranteed that no soft transaction can cause the system to fail.

2.2.4 Admission Control

Controlling the admission of transactions in a real-time database has been investigated in work by Haritsa et. al. [HCL90] and Goya et al [GHS+95]. In recent work by Bestavros and Nagy [BN97] an admission controller receives submitted transactions and determines whether to admit or reject transactions. Transactions that are admitted to the scheduler are then guaranteed to meet their deadlines. The decision to reject a transaction is based upon a feedback mechanism that takes into consideration current demands for resources in the system.

2.3 Dependability Concepts

Dependability is an important concept in real-time computing. A real-time system is required to be dependable. For example, it would be difficult to sell an automatic car control system that is not dependable. Laprie et. al. [Lap94] define dependability as: “that property of a computer system such that reliance can justifiably be placed on the service that it delivers”. According to Laprie et. al. [Lap94] dependability has a number of attributes such as:

- *Availability* is concerned with readiness of usage, e.g., a car that always start has high availability.

- *Reliability* is concerned with continuity of service, e.g., a reliable car is does not frequently break down.

- Safety is measured by the non-occurrence of catastrophic consequences on the environment, e.g., a car is not safe it its breaks consistently fail to operate.

- Maintainability is concerned with the aptitude to undergo repairs and evolution, e.g., the ability to repair or change parts of a car.

Laprie et. al. [Lap94], define a fault as the hypothesized cause of an error, where an error is an incorrect state of the system which can lead to a failure. A failure has occurred when the system has deviated from its specification. The presence of a fault can be shown through the occurrence of failures or errors.

To provide for dependability a number of methods can be applied. These can be classified into [Lap94]:

- **Fault prevention**: how to prevent faults occurrence or introduction.
- **Fault tolerance**: how to provide service in spite of faults.
- **Fault removal**: how to reduce the presence of faults (see section 2.4).
- **Fault forecasting**: how to estimate present number, future incidents and future consequences of faults.

Fault prevention, i.e., preventing faults from occurring is not a realistic option. No software engineering method can eliminate all faults. Therefore, methods to tolerate faults are needed. A fault tolerant system attempts to detect and correct errors before the become effective.

Fault removal in done in three steps: verification, diagnosis, and correction. Verification determines the correctness of the system with respect to its specification. Static and dynamic verification techniques are used. Static verification involves checking the system without executing it. Dynamic verification checks the system while executing it (see section 2.4). If it is determined during verification that the system does not match its specification the problem is diagnosed and corrected.

According to Laprie et. al. [Lap94], fault forecasting has two aspects:

- Qualitative forecasting aims at i) identifying, classifying, and ordering failure modes or ii) identifying event combinations leading to undesired events.
- Quantitative forecasting evaluates in terms of probabilities attributes of dependability.

2.4 Software Testing

Testing is a verification technique where a program is executed to check if it deviates from its specification. According to Beizer [Bei90] and Sommerville [Som92], the aim of testing is to discover faults in software and testing can only show the presence of faults, not their absence. Testing is one of the most widely known and applied verification technique. It is also a widely applicable technique because, in general, systems of all sizes and shapes can be tested. Testing is never completed, in the sense that exhaustive testing is practically impossible. The goal is to employ techniques that maximize the likelihood of revealing faults with the amount of testing that can be applied in each case. Still the effort associated with testing is very large, and it has been reported by Myers [Mye79], Hetzel [Het88] and Beizer [Bei90], that testing often consumes at least 50% of the total project cost.

2.4.1 Levels of Testing

Beizer [Bei90] defines three levels of testing: unit/component testing, integration testing, and system testing.

- Unit testing: A unit is the smallest testable part of a systems and is tested to see if it violates its functional specification and/or the intended design structure.

- Component testing: A component consists of one or more units and is tested with respect to the same criteria as units.

- Integration testing: The aim of integration testing is to look for faults in the interaction between components. Even if each component has been individually tested, the integration of several components may introduce inconsistencies among the components.

- System testing: System testing executes the entire system and includes testing factors such as performance, stress, security and recovery.

2.4.2 Testing Techniques

According to Beizer [Bei90], testing can either be based on functional or structural information. Structural testing[3] uses knowledge about the structure of the program to create tests. Conversely, functional testing views the software as a black-box, where the internals of the software are not visible to the tester. Structural testing is most applicable in unit and component testing, while functional testing is more appropriate for integration and system testing. Various testing techniques based structural or functional information have been suggested, e.g., transaction flow testing, syntax testing, state testing, and path testing [Bei90].

2.4.2 Testing Strategies

Different testing strategies have been suggested. The most common are top-down and bottom-up testing. According to Sommerville [Som92] top-down testing starts at the sub-system[4] level. The components are represented by stubs which implement interfaces to components. Once sub-system testing is completed components are tested and each

[3] Also known as white-box testing
[4] Same as integration testing in Beizer's test levels

unit is represented by a stub. Finally all stubs are replaced by actual code and the system is tested.

Bottom-up testing is the converse of top-down testing. It starts testing at the units and proceeds upwards until the complete system is tested.

2.4.3 Definitions

Three central concepts in testing are test effort, test coverage and testability. The concepts of a fault, failure, and error are also fundamental in testing.

Test effort refers to the effort associated with testing a particular test object, where a test object is the software component of interest. It includes designing test cases, instrumenting the test object, executing tests, and analyzing the results.

Test coverage is a measurement of how well the generated test cases capture possible real-world scenarios. To achieve full test coverage requires testing the complete input space. For real-time systems the input space has three dimensions: possible input values, possible states of the system, and time. Test effort is related to test coverage in the sense that ensuring greater test coverage increases the test effort because more test cases are needed.

Testability is a measurement on the ease with which we can test a particular test object. According to Schütz [Sch93], in the context of real-time systems the most influential factors on testability are: test effort, observability, and reproducibility (see section 2.5.1).

2.5 Testing Real-Time Systems

Real-Time systems are often employed in safety-critical environments, such as flight-control, life-support systems, and nuclear power plants where a failure may have catastrophic consequences. The emphasis on testing under such circumstances is increased. At the same time testing real-time systems is more difficult than testing non real-time systems. Instead of only testing the functional correctness of a program it is also necessary to test for timeliness, i.e., ensuring that the system meets its deadlines.

2.5.1 Observability and Reproducibility

When executing tests it is necessary to observe the behavior of the system. Observing the output of the system is not sufficient. According to Schütz [Sch93], we also need to know how the system reached the output and when. Two common ways to observe intermediate results are to instrument the system with output statements at appropriate points in the code or to use an interactive debugger. Neither of these are applicable for real-time systems since they have an effect on when events or actions happen in the system and thus change its behavior. According to Mellin [Mel96, Mel95], one way to solve the observability problem, also known as the Probe Effect [Gai85], is to use an active real-time database system.

According to Schütz [Sch93] reproducibility is another significant problem that needs to be dealt with when testing real-time systems. Reproducibility is the ability to reproduce the behavior of the system when presented with a particular input sequence. Reproducibility is necessary to enable regression testing, which means running the same test cases again to see if changes made in the software have had the desired effect and that no new faults have been introduced as a result of these changes. Being able to observe how a system reacts to an input sequence is necessary when testing a system. In contrast, reproducibility is only a highly desirable attribute because it facilitates regression testing.

2.5.2 Impact of Design Paradigm on Testability

According to Schütz [Sch93] and Sommerville [Som92], the chosen design paradigm of a real-time system has a large effect on testability, i.e. how easily software can be tested. Schütz states that time-triggered systems are inherently easier to test than event-triggered system due to the nature of their architecture. The contributing factors are:

- The coupling between input values and time is loose, because we have a discrete time frame, consisting of the observation points.

- There is only one synchronization sequence, i.e. a sequence of synchronization events that uniquely determines the results of a concurrent program execution. Hence, all the effort can be focused on this synchronization sequence.

- All computations must finish within the interval of two action points and within this interval they are executed in some fixed order. As a result no preemption is required.

- The order of inputs between two observation points is not important

- Access to time can be made deterministic.

2.5.3 Test Effort for Time-Triggered Real-Time Systems

Schütz [Sch93] compares the test effort required for an event-triggered real-time system to that of a time-triggered. The test effort is considered proportional to the number of states that the environment can be perceived in during an action interval g_a. If, during this interval, n independent events can arrive and the environment can be observed s times the number of perceivable environment states is:

$$\sum_{k=0}^{n} \binom{n}{k} s^k = \sum_{k=0}^{n} \binom{n}{k} s^k 1^{n-k} = (s+1)^n \qquad (1)$$

In time-triggered systems the environment is only observed once during each interval and thus s=1. For an event-triggered system the maximum number of observations during g_a is defined as [Sch93] :

$$S_{E\max} = \frac{g_a}{\max(g_{instr}, g_{obs})} \qquad (2)$$

Where g_{instr} is the granularity of machine instruction execution and g_{obs} the time needed to handle a single observation.

Schütz notes that using S_{Emax} with (1) gives us a lower bound for the number of observable environmental states in an event-triggered system. The lower bound given by Schütz for event-triggered systems, does not take into account when inputs arrive simultaneously or handle preemptions.

2.5.4 Test Effort for Event-Triggered Real-Time Systems

In the previous section a test effort lower bound for event-triggered systems was discussed. An upper bound for event-triggered systems is presented by Mellin [Mel96]. To formulate this upper bound Mellin introduces two constraints on behavior of the system. The first constraint defines a maximum on the number of times a transaction can be preempted. The second constraint defines a maximum on the number of concurrently executing transaction instances triggered by the same event type. As discussed in section 2.2, Mellin [Mel96] also shows how the presence of an active database system can improve the testability of applications.

The upper bound, as defined by Mellin [Mel96], for the test effort of event-triggered real-time systems is based on a number of assumptions. Firstly, deadlock prevention is used to eliminate the possibility of deadlocks in the system. This can be achieved by requiring all resources used by a transaction being allocated before the transaction executes. Secondly, at any time all transaction instances are assumed to have a unique priority. Thirdly, only a single resource can be locked at a time. Fourthly, the contents of the entire database are kept in main-memory. This eliminated all I/O related blocking. Fifthly, transactions acquire exclusive locks to resources.

In the work by Mellin [Mel96] it is also suggested that the points in time where the system can observe the environment are constrained. At each observation point the

computing element is presented with events that have occurred since the last observation. This creates a looser coupling between time and value. Moreover, a coarser observation interval reduces the test effort.

The upper bound as defined by Mellin [Mel96] is formulated by combining the number of perceivable environment states (see section 2.4.3) with the number of potential blockings represented in (4) BSTAT and potential preemptions represented in (5) PSTAT. The number of potential blockings presented in (4) BSTAT is based on finding all combinations of two or more transactions[5] out of a total of n*q executing transactions, where n is the number of event types and q the maximum number of concurrently executing transactions triggered by the same event type. Only combinations of blockings are included because, at each point in time, all transaction instance are assumed to have a unique priority. The number of potentially preempted transactions presented in (5) PSTAT is found by considering all possible combinations of preempted transactions out of n*q transactions when each transaction can be preempted at most p times. Together BSTAT and PSTAT represent all possible previous states of transactions in the system. ESTAT, as formulated by Schütz [Sch93] represents the number of states that the system can perceive the environment in. At each observation point the number of future states is based on all possible previous states multiplied by all perceivable environmental states. Therefore, to cover all possible cases each environmental state from ESTAT must be combined with all possible previous states of transactions in the system. This results in (3) FSTAT where possible future events are multiplied with all possible previous states represented by BSTAT*PSTAT. As pointed out by Mellin [Mel96], the upper bound represented in FSTAT is conservative and pessimistic because multiplying BSTAT*PSTAT results in previous states that are not possible, i.e., by definition a blocked transaction can not be preempted and vice versa.

$$FSTAT(s,n,p,q) = ESTAT(s,n) * BSTAT(n,q) * PSTAT(n,p,q) \quad (3)$$

[5] Tasks and transactions are equivalent in this discussion.

$$BSTAT(n,q) = \sum_{k=2}^{nq} \binom{nq}{k} 1^k 1^{n-k} \Leftrightarrow BSTAT(n,q) = \sum_{k=0}^{nq} \binom{nq}{k} 1^k 1^{n-k} - \sum_{k=0}^{1} \binom{nq}{k} \Leftrightarrow \quad (4)$$
$$BSTAT(n,q) = 2^{nq} - (nq+1)$$

$$PSTAT(n,p,q) = \left(\sum_{k=0}^{q+1} p^k\right)^n = \begin{cases} \left(\frac{p^{q+2}-1}{p-1}\right)^n & when\ p \neq 1 \\ (q+2)^n & when\ p = 1 \end{cases} \quad (5)$$

$$FSTAT(s,n,p,q) = (s+1)^n * \left(2^{n*q} - (n*q+1)\right) * \begin{cases} \left(\frac{p^{q+2}-1}{p-1}\right)^n & when\ p \neq 1 \\ (q+2)^n & when\ p = 1 \end{cases} \quad (6)$$

2.7 The DeeDS Architecture

In this work it is assumed that applications use an active real-time database system which has similar properties as DeeDS[AHEM95, AHE+96], a **D**istributed Activ**e** Real-Tim**e** **D**atabase **S**ystem developed by the distributed real-time systems group at the University of Skövde. DeeDS combines active behavior with a distributed real-time database system. The database in DeeDS is main-memory based (see section 2.2.2), and the reactive behavior is modeled with event-condition-action (ECA) rules (see section 2.2.3). Scheduling is dynamic (see section 2.1.2), and transactions are either periodic or sporadic, with soft or hard deadlines. In the DeeDS architecture, application related functions are separated from critical system services. The former consist of rule management, an object storage, and storage management. The latter includes include scheduling and event monitoring.

Chapter 3

Problem Description

This chapter presents the problems faced and discusses the purpose of this work. A model for real-time systems that are constrained to improve testability is presented.

3.1 Motivation

This work is mainly motivated by two factors. Firstly, methods to make event-triggered systems more testable are essential if the required level of confidence in correctness of event-triggered real-time systems is to be achieved. Secondly, knowledge about feasibility of such methods is needed to determine if and how they can be applied.

The use of an active database to handle observability and an approach to reduce the test effort has been proposed by Mellin [Mel96] (see section 2.5.4). The first motivation for this work is concerned with validating and refining the work done by Mellin. The second motivation is concerned with investigating constraints in Mellin's work to determine the feasibility of applying them to improve testability of event-triggered real-time systems.

3.2 Problem Delineation

As discussed in section 2.7 this work assumes the tested application to be based on an underlying active real-time database system which similar properties as DeeDS [AHEM95,AHE+96] (see section 2.7). The distribution aspect is not considered in this work, since a solution for a single node must first be found. Certainly, if a solution does not exist for a single node it does not exist for a distributed real-time system.

Testing can be performed on different levels (see section 2.3). In this work the focus is on system testing although the results can apply to lower levels such as integration testing. Moreover, this work focuses on testing applications for timeliness.

According to Schütz[Sch93], the most important aspects of testability for real-time systems are test effort, observability, and reproducibility (see sections 2.4.1 and 2.4.3). This work focuses on the test effort, because reducing the test effort is essential if achieving full test coverage when testing event-triggered systems should become feasible. Achieving observability is necessary for testing of any kind. It can be obtained with an active real-time database system which has a predictable built-in event monitor, and in this dissertation it is assumed that such a system exists. Reproducibility becomes important only if the test effort and observability can successfully be handled.

3.3 Problem Description

In the previous section it was stated that the focus of this work is on reducing the test effort for event-triggered real-time systems. As discussed in section 2.5.4, to define an upper bound on the test effort for event-triggered real-time systems their behavior must be constrained. The reason is that an unconstrained event-triggered real-time system has a potentially very large number of synchronization sequences, i.e., sequences of synchronization events that uniquely determine the results of a concurrent program execution. Therefore, achieving full test coverage is not feasible unless we can control the number of possible synchronization sequences. As discussed in section 2.5.4, Mellin [Mel96] defines an upper bound on the test effort for event-triggered real-time systems by applying constraints on system behavior. The correctness of formulas that define the upper bound is not completely validated in the work by Mellin. Therefore, there is room for refinement and possibly corrections of the formulas. Moreover, assuming that only a single resource can be locked at a time is not realistic (see section 2.5.4).

The upper bound on the test effort for event-triggered real-time systems is high compared to time-triggered systems. Therefore, it is important to investigate ways to reduce the test effort further

The effects of applying the constraints proposed by Mellin [Mel96] have not been fully investigated. Constraining the number of times a transaction can be preempted is bound to have an effect on a real-time system. The same applies to constraining the number of concurrently executing transaction instances triggered by the same event type and observation points. When investigating the effects of these constraints various real-time properties as well as system services need consideration. One of the problems related to investigation of effects is the selection of properties. Properties of real-time systems are, e.g., predictability, efficiency, dependability, and extensibility. Likewise, the selection of system services such as the scheduling, event monitoring or rule management needs to be addressed.

The introduction of constraints to improve testability requires special attention during the design phase. Selection of constraint values is done during design. The selected constraint values must result in a feasible application, e.g., the length of an observation interval must be such that it is still possible to meet all deadlines.

Enforcement of constraints is an important step to realize the usage of constraints for improving testability of event-triggered real-time systems. The constraints improve the testability of applications by reducing the test effort. If the constraints are violated during testing the results of the test are no longer valid. The reason for this is that the upper bound defines the number of tests that have to be run in order to fully test the system. A violation of the constraints, e.g., preempting a task that has reached maximum preemptions, means that no such guarantee can be given because the execution of the system is no longer inside the scope defined by the upper bound. It is the responsibility of the active real-time database system to make sure that the constraints are not violated or to inform the user when they are. Three general methods for enforcement are prevention, avoidance, and detection and recovery. The applicability of these as well as mechanisms to realize enforcement need investigation.

3.4 Model of Constrained Event-Triggered Real-Time Systems

In this section a model for event-triggered real-time systems that use constraints to improve testability is suggested. This model is the basis for all discussion in remaining

chapters of this dissertation. To increase testability of a real-time system, using the methods described in this work, a number of assumptions must hold. These assumptions are divided into four sub-models: a transaction model, an execution model, a resource model, and a system level testing model. A transaction model is defined to specify properties that transactions must have. An execution model defines how transactions are executed in the system. A resource model defines how resources are allocated and used by transactions. Finally, a model for system level testing is presented to clarify how a test case is constructed and executed in order to test timeliness.

3.4.1 Transaction Model

The following transaction model is assumed:

T1) Each transaction has a hard or soft deadline.

T2) Transactions are preemptable from the processor. Being able to remove a transaction from the processor is a requirement for preemptive scheduling which is assumed in this work.

T3) Nested transactions are not allowed, because they are a source of unpredictability.

T4) Transactions are either periodic or sporadic.

T5) Worst-case execution time, deadline, earliest release time, and minimum interarrival rate are known for each transaction with a hard deadline.

T6) The priority of transaction instances is unique. Requiring unique priorities reduces the test effort because fewer synchronization sequences are possible.

T7) A transaction instance is triggered by a single event type which is either critical or non-critical.

T8) The triggering event type of a transaction instances in assumed to be known, because knowledge of the triggering event type makes it possible to discard transaction instances according to event semantics (see section 5.3)

3.4.2 Execution Model

The following execution model is assumed:

E1) A dynamic preemptive scheduling algorithms is used.

E2) By using off-line analysis all hard transactions are guaranteed to meet their deadlines. If worst-case execution time for soft transactions is not known no guarantees are given for soft transactions.

E3) An executing soft transaction, that has been preempted maximum number of times, is always aborted if a hard transaction arrives.

E4) A soft transaction, that is aborted by the system in favor of a hard transaction, is assumed to be resubmitted if necessary. The nature of soft transactions is such that it is useful to eventually complete them. It is for example not of paramount importance to execute a transaction that controls air temperature in a car. But, the driver would eventually like the transaction to execute and adjust the temperature.

E5) The time to abort a soft transaction is bounded. The time to abort soft transactions must be bounded in order to guarantee timeliness of hard transactions (see section 5.1).

3.4.3 Resource Model

A resource model defines how resources are allocated, shared and accessed. A refined resource model based on the used in Mellin's work is assumed. The difference is that

more than one resource can be locked at a time. This results in the following resource model:

R1) All resources are exclusively locked, because this simplifies the resource model.

R2) A transaction must acquire locks on all required resources before execution. Requiring transactions to obtain all locks before execution prevents deadlocks from occurring.

R3) If r is the number of resources at most r resources can be locked at any time. Allowing all transactions to access all resources is chosen because of its simplicity. Restricting access to resources would on the other hand reduce the number of potential blockings.

R4) A resource in this work is a database object.

3.4.4 System Level Testing Model

As discussed in section 2.4 system level testing is concerned with testing the entire system. System level testing assumes that confidence in correctness of units, components, and integration of components has been attained at each respective test level. The importance of timeliness (see section 2.1, p.11) in a real-time system makes it of paramount interest in system testing. Timeliness, of a real-time system, can only be tested once the entire system has been implemented.

In section 2.4.1 test effort was defined as consisting of the effort in designing test cases, instrumenting the test object, executing tests, and analyzing the results. To fully test a system with respect to timeliness requires generating all possible test cases and executing each case to see if the system meets its deadlines. As already mentioned in this section an unconstrained system requires prohibitively many test cases to achieve full test coverage. A more feasible approach is to use the constraints suggested by Mellin [Mel96], which bound the number of required test cases.

It is important to clarify what a test case consist of in this work and how testing is done. Figure 3.1 shows the information included in each test case. A tests case contains information about state of the system and events that occur in the test. The state of the system contains information about the number of times each transaction has been preempted and currently blocked transactions. The state of the system represents the past, i.e., the state before testing starts. The point in time where testing starts is marked in Figure 3.1 as *now*. The objective of testing, in the context of timeliness, is to check if the system meets its deadlines after *now*. Executing a test case requires the system to be in the state specified by the test case. Once the system is in this state execution is started and events in the test case are presented to the system. Execution of transaction is monitored to see if any deadlines are missed.

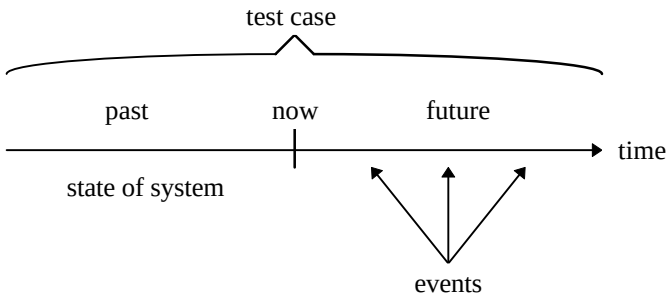


Figure 3.1. Test case.

In appendix D an example that illustrates how test cases are generated is given.

3.5 Summary of Identified Problems

This section presents a brief summary of the problems identified in the previous section, and which will be addressed in this dissertation.

- Validation of upper bound for test effort of event-triggered real-time systems is required to investigate need for corrections or refinements to the original

formulation states by Mellin. The test effort upper bound as defined by Mellin is high. Ways to reduce the test effort further are needed.

- Little investigation of effects that constraints that improve testability have on applications or the active real-time database systems has been performed. This is necessary to determine the feasibility of applying these constraints.

- Designing applications that use the constraints to improve testability requires special consideration. Guidelines are needed for selecting constraint values that result in a feasible application.

- To realize the usage of constraints for improving testability of applications enforcement of the constraints must be done by the active real-time database system. This requires investigating applicability of enforcement methods, and way to realize the enforcement in an active real-time database system.

Chapter 4

Validation and Refinement of Constraints

This chapter describes the validation and refinement of existing constraints that improve testability of applications in active real-time database system environments. In section 4.1 problems with the original formula describing an upper bound on the test effort for event-triggered real-time systems are identified. Section 4.2 presents corrections to the existing formulas. Section 4.3 presents refinements to the existing formulas that provide a more accurate upper bound on the test effort and further reduce the test effort by only testing hard transactions completely.

4.1 Validation of Upper Bound Test Effort

This section discusses problems with previous work on constraining event-triggered real-time systems to improve testability. As discussed in section 3.3 the number of possible states for an unconstrained event-triggered real-time system is prohibitively large because of the enormous number of possible synchronization sequences [Sch93]. Therefore, in order to formulate an upper bound on the test effort for event-triggered real-time systems, certain constraints must be placed on system behavior. This is done in work by Mellin (see section 2.5.4). The following problems with the upper bound presented in formulas (3)-(6) in section 2.5.4 have been identified:

1) Formula for preemption states (5) calculates fewer states than there actually are.
2) Event types and event instances are not distinguished.
3) Formula for blocked states (4) only covers cases where a single resource is locked.

The first two problems are due to a minor error in the formulation and require modest changes to the original formulae. The third problem is more difficult to solve and requires a new formula for potential blockings. In the section 4.2 corrections that solve the first two problems are presented. In section 4.3 a new formula for blocked states is presented.

4.2 Correction of Upper Bound Test Effort Formulas

In his work Mellin [Mel96] separates between event types and event instances. Each event type can trigger a number of transaction instances. This separation is not made in the formulas because the variable n is used to express events types as well as possible independent events in ESTAT. To clarify the distinction between types and instances a new variable t is introduced. In (7) a revised FSTAT that separates between event types and event instances is presented.

$$FSTAT'(s,n,p,q,t) = (s+1)^n * (2^{t*q} - (t*q+1)) * \begin{cases} \left(\frac{p^{q+2}-1}{p-1}\right)^t & when\ p \neq 1 \\ (q+2)^t & when\ p=1 \end{cases} \quad (7)$$

Further analysis of the formulas presented by Mellin [Mel96] reveals that PSTAT calculates fewer preemption states than there actually are. The reason is that the original formula does not contain cases where a transaction is not preempted. To illustrate this point consider a system where three transaction instances of the same type exist and each transaction can be preempted at most once. In a given interval each of the three transactions can either execute or not. If a transaction executes it has been preempted once or not at all. This means that cases where no preemption occurs must be included and as a result referring to preemption states is not precise. A more appropriate term,

that will be used in this work, is that of execution states. To calculate the correct number of execution states the term p^k in PSTAT (5) is replaced by $(p+1)^k$ in PSTAT´ (8).

Another error in the formula is that the number of concurrently executing transaction instances triggered by the same event type are counted from zero instead of one. Using the original formula this would mean that if zero transaction instances are allowed to be triggered by the same event type then the number of possible execution states are two. But in a system without transactions no execution states are possible. To correct this error the sum from k=0 to q+1 in PSTAT (5) is replaced with a sum from k=0 to q in PSTAT´ (8). Further, PSTAT´ requires that a transaction can be preempted at least once. If this is not the case then preemptive scheduling is no longer an option, but most event-triggered real-time systems require preemptive scheduling. A verification of PSTAT´ (8) is presented in Appendix B.1.

A revised formula for the upper bound on the test effort for event-trigged real-time systems is presented in FSTAT´´ (9). It separates between event types and instances, includes a corrected formula for execution states, and for completeness formulates the case when no blocking can occur, i.e., when system only has a single transaction.

$$PSTAT'(p,q,t) = \left(\sum_{k=0}^{q}(p+1)^k\right)^t = \left(\frac{(p+1)^{q+1}-1}{p}\right)^t \quad when\ p>0 \quad (8)$$

$$FSTAT''(n,s,p,q,t) = (s+1)^n * (2^{q*t} - (q*t+1)) * \left(\frac{(p+1)^{q+1}-1}{p}\right)^t$$

$$when\ \ q*t \geq 2 \ \ \wedge \ \ p>0 \qquad (9)$$

$$FSTAT''(s,n,p,q,r) = (s+1)^n * \left(\frac{(p+1)^{q+1}-1}{p}\right)^t \quad when\ \ t*q=1 \ \ \wedge \ \ p>0$$

4.3 Refinement of Upper Bound Test Effort Formulas

This section presents refinements to the formulation of the upper bound on the test effort for event-triggered real-time systems. Two refinements are discussed: a refinement of the formula for potential blockings, and a refinement on the number of observable environmental states.

4.3.1 Refined Formula for Potential Blockings

To discuss refinement of the formula for blocked states a notation for representing potential blockings is needed. The notation x [y is used to represent the situation when transaction x blocks transaction y. The situation where x blocks more than one transaction is represented by x [y_1, y_2, y_3, ..., y_n. Transaction x and all transaction blocked by x form a blocked chain. Several such chains can exist at the same time. A conjunction operator ^ connects two or more chains. The situation where x blocks y and z blocks q is represented by x [y ^ z [q.

In the work by Mellin [Mel96], the formula representing potential blockings of transactions is based on the sum of all combinations of two or more transactions. This only covers the case situation where a single resource is locked. If two or more resources are locked at the same time the number of potential blockings increases. For example, assume we have a system with four transactions a,b,c, and d with increasing priorities, a having the lowest and d the highest. Further, the system has two exclusively locked resources, i.e., a resource can only be locked by a single transaction. If only a single resource can be locked at a time potential blockings are:

No.	Blockings
1	a [b
2	a [c
3	a [d
4	b [c
5	b [d

6	c [d
7	a [b,c
8	a [c,d
9	a [b,d
10	b [c,d
11	a [b,c,d

Table 4.1. Potential blockings for a single resource

In contrast, if more that one resource can be locked at a time the potential blockings are:

Nr.	Blockings
1	a [b
2	a [c
3	a [d
4	b [c
5	b [d
6	c [d
7	a [b,c
8	a [c,d
9	a [b,d
10	b [c,d
11	a [b,c,d
12	a [b ^ c [d
13	a [c ^ b [d
14	a [d ^ b [c

Table 4.2. Potential blockings for two resources.

The result of more than one resource being locked is that instead of single chains of blocked transactions several chains can be formed (cases 12-14 in table 2). The length of a chain is determined by the number of transactions that make up the chain. The minimum length of a chain is two and the maximum length is equal to the maximum number of transactions.

The problem of finding a way to calculate the number of potential blockings, when more than one resource can be locked at a time, can be split into two sub-problems. The first sub-problem is to find a formula for calculating the number of potential blocked states resulting from a single group of chains. A group of chains is represented by [x_1, x_2, x_3,..., x_n] where x_1...x_n are chains of length x. For example [5,4,2] represents a situation such as: a [b,c,d,e ^ f [g,h,i ^ j [k. The second sub-problem is to develop an algorithm that creates all possible groups of chains for a given number of transactions and resources.

Calculating potential blockings is done in two steps. First, all possible groups of chains for the given number of transactions and resources are created (see algorithm 1, p.49). Second, potential blockings for each group of chains are summed. The result is the number of potential blockings for the given number of transactions and resources.

Potential blockings for one group of chains

Consider the following example where a system has nine transactions and three resources. One possible group of chains is [4,3,2]. The first chain, of length four, can be constructed in as many ways as there are four-combinations out of a set of nine elements. All these chains of length four can then be combined with all combinations of chains of length three out of the remaining five elements. Finally, all chains of length four and three can be combined with the single chain of two elements out of the remaining two elements. In BSTAT´ (10) the number of potential blockings for a group of chains where each chain has a unique length[6] is formulated. In BSTAT´, t is the number of event types, q the maximum number of concurrently executing transaction instances triggered by the same event type, and C a set containing chains.

[6] Each chain in [4,3,2] has a unique length. In [4,3,3] two chains have the same length.

$$BSTAT'(q,t,C) = \binom{qt}{c_1} * \binom{qt-c_1}{c_2} * \binom{qt-(c1+c_2)}{c_3} * \ldots * \binom{qt-(c_1+c_2+\ldots+c_{r-1})}{c_r} =$$

$$\prod_{k=0}^{r-2} \binom{q*t-\sum_{l=0}^{k} c_l}{c_{k+1}} \quad where \quad \forall i : 1 \le i \le r \mid c_i \in C \quad \wedge \quad \left(\sum_{i=1}^{r} c_i\right) \le q*t \quad \wedge c_0 = 0 \qquad (10)$$

$$\wedge \quad \forall i, j : 1 \le i, j \le r \quad \wedge \quad i \ne j \quad \mid \quad c_i \ne c_j$$

If the length of each chain in a group is the same then BSTAT´ includes redundant cases. For example if we have four transactions and two resources then for the group [2,2] the number of states given by BSTAT´ is:

$$BSTAT(4,1,\{0,2,2\}) = \prod_{k=0}^{3-2} \binom{4*1-\sum_{l=0}^{k} c_l}{c_{k+1}} = \binom{4}{2} * \binom{2}{2} = 6$$

Which corresponds for example with the following scenario:

Nr.	Blockings
1	a [b ^ c [d
2	a [c ^ b [d
3	a [d ^ b [c
4	b [c ^ a [d
5	b [d ^ a [c
6	c [d ^ a [b

Table 4.3. Redundant blockings.

Note that the three last chains only mirror the previous three and the real number of potential blockings is only three instead of six. To eliminate these redundant cases changes to BSTAT´ (10) are required. The resulting formula is presented in BSTAT´´ (11). In BSTAT´´ the variables t, q, and are the same as for BSTAT´´. The variable C from BSTAT´ is replaced by C´ which is a bag[7] containing chains. A new variable O is introduced. O is a bag containing the number of occurrences of each element in C´. For example, if C contains the chains [5,4,4,3,3,3,2] O contains [1,2,3,1].

$$BSTAT''(q,t,C,O) = \frac{\binom{qt}{c_1} * \binom{qt-c_1}{c_2} * \binom{qt-(c_1+c_2)}{c_3} * \ldots * \binom{qt-(c_1+c_2+\ldots+c_{r-1})}{c_r}}{o_1! * o_2! * o_3! * \ldots * o_n!} =$$

$$\frac{\prod_{k=0}^{r-2} \binom{q*t-\sum_{l=0}^{k} c_l}{c_{k+1}}}{\prod_{m=1}^{n} o_m!} \quad where \ \forall i: 1 \le i \le r \mid c_i \in C \ \wedge c_0 = 0 \wedge \left(\sum_{i=1}^{r} c_i\right) \le q*t \qquad (11)$$

$$\wedge \ \forall j: 1 \le j \le n \mid o_i \in O \ \wedge \ \forall i, j: 1 \le i, j \le r \ \wedge \ i \ne j \ \mid \ c_i \ne c_j$$

Using the same example the number of potential blockings according to BSTAT´´ (11) is:

$$BSTAT''(4,1,\{0,2,2\},\{2\}) = \frac{\prod_{k=0}^{3-2} \binom{4*1-\sum_{l=0}^{k} c_l}{c_{k+1}}}{\prod_{m=1}^{1} o_m!} = \frac{\binom{4-0}{2} * \binom{4-2}{2}}{2!} = \frac{\binom{4}{2} * \binom{2}{2}}{2!} = \frac{6}{2} = 3$$

[7] The elements in a bag are the same as in a group. The difference between a bag and a set is that a bag can contain the same element more than once.

An informal verification of BSTAT´´ is given in Appendix C. Using BSTAT´´ instead of BSTAT in FSTAT´ gives FSTAT´´´ (12).

$$FSTAT'''(n,s,p,q,t,C,O) = (s+1)^n * \frac{\prod_{k=0}^{r-2} \binom{q*t - \sum_{l=0}^{k} c_l}{c_{k+1}}}{\prod_{m=1}^{n} o_m!} * \left(\frac{(p+1)^{q+1}-1}{p}\right)^t$$

$$when \quad q*t \geq 2 \quad \wedge \quad p > 0 \tag{12}$$

$$FSTAT'''(s,n,p,q,r) = (s+1)^n * \left(\frac{(p+1)^{q+1}-1}{p}\right)^t \quad when \quad t*q = 1 \quad and \quad p > 0$$

Algorithm for constructing groups of chains

As mentioned above the second sub-problem in calculating the number of potential blocked states is to devise an algorithm that creates all possible sets of chains when we have n transactions and r resources. A convenient way of looking at this problem is to express it as the problem of finding all sums of one or more numbers that are larger than one and less than or equal to q*t (number of transactions). For example there are twelve different sums that are larger than one and less than or equal to seven (2, 3, 4, 5, 6, 7, 2+2, 3+2, 3+3, 4+2, 4+3, 5+2) if at most two elements (resources) can be used. If three elements can be used fourteen sums are possible (2, 3, 4, 5, 6, 7, 2+2, 3+2, 3+3, 4+2, 4+3, 5+2, 2+2+2, 3+2+2). In these examples the sums cover all possible chains in a system with seven transactions and two and three resources respectively. An algorithm to solve this problem is shown below.

procedure SumChains(N,R: positive integers) {N is max sum and R is max elements}

1: **for** i := 1 to R **do**

2: **begin**

3: **for** j := 1 to i {start with 2+2+2...+2} **do** c_j := 2 { c is an array of size i }

4: **if** sum of elements in c <= N **then** add c to list L

5: **while** c_0 <= N - (i-1)*2 **do**

6: **begin** {increase the first occurrence of lowest element by one}

7: m := c_i

8: f := position of first occurrence of m in c

9: c_f := c_f + 1

10: **for** k := f+1 to i {set all elements from f to 2} **do** c_k=2

11: **if** sum of elements in c <= N **then** add c to list L

12: **end**

end { L contains all chains of sums, of size r or less, that are less or equal to r }

Algorithm 1. An algorithm to create all possible chains in a system with N transactions and R resources.

In appendix G an example of how this algorithm works is given.

4.3.2 Reducing the Test Effort Further

The test effort upper bound for event-triggered systems is high. Take for example a very simply system with the following properties: three independent events (n) can arrive during an observation interval; a transaction can be preempted (p) twice; three event types (t) exist and only one transaction instance (q) triggered by each of these types is allowed to execute at any time; the transactions require access to one resource. If three observations (s) are performed then according to (9) the upper bound is:

$$S(8,5,2,2,2) = (3+1)^3 * (2^3 - (3+1)) * (\frac{(2+1)^{1+1} - 1}{2})^3 = 4^3 * (2^3 - 4) * 4^3$$
$$= 64 * 4 * 64 = 16384$$

To fully test this simple system requires designing and execution over sixteen thousand test cases.

Two ways to reduce the test effort are discussed in this section. The first approach is to remove those environmental states where no hard transactions occur during the observed interval. In other words, test cases for situations where only soft transactions arrive are not included because the primary interest is in testing timeliness of hard transactions. In (12) the resulting formula for ESTAT´ is given.

$$ESTAT'(s,n,h) = (s+1)^n - ((s+1)^{n-h} - 1) \qquad (12)$$

A new variable h is introduced which represents the number critical events out of the n events that arrive. The difference between the original ESTAT and ESTAT´ is shown in Table 1.

ESTAT	ESTAT'	s	n	h	%
729	703	2	6	3	3.57
2187	2107	2	7	3	3.65
6561	6319	2	8	3	3.69
4096	4033	3	6	3	1,53
16384	15361	3	7	3	1,56
65536	64513	3	8	3	1.56
15625	15501	4	6	3	0.79
78125	77501	4	7	3	0.80
390625	387501	4	8	3	0.80
729	242	2	6	1	33%

16384	12288	3	7	1	25%
390625	312500	4	8	1	20%

Table 4.4. Difference between ESTAT and ESTAT´

The reduction achieved by using ESTAT´ is related to the ratio between critical and non critical events. As the number of critical events decreases the number of states also decreases. This can be seen by examining what happens when h approaches zero.

$$\lim_{h->0} (s+1)^n - ((s+1)^{n-h} - 1) = 1$$

From the examples given in table 4.4 It is obvious that this approach does not have any considerable impact on the upper bound when the number of critical and non critical events in similar. On the other hand, the cases that are removed are not of importance and therefore (12) can safely be used instead ESTAT in the original formula (6). In appendix A a comparison and verification of ESTAT and ESTAT´´ is done.

The second approach is to test only a subset of the execution states. All execution states of transaction instances that are triggered by a critical event type are included. For transaction instances triggered by a non-critical event type only certain execution states are included. The removed states are those where transactions have not been preempted maximum number of times. The reason is that these do not effect the execution of hard transactions. The situation where a soft transaction is not preempted can be considered as representative for situations where that transaction is preempted less than the allowed maximum. When a soft transaction has reached maximum preemptions a separate test case is needed because now the soft transaction can potentially delay execution of a hard transaction. The resulting formula for execution states is presented in PSTAT´´ (13). A new variable w is introduced to represent the number of critical event types. In PSTAT´´ the first term $\left(\sum_{k=0}^{q} (p+1)^k \right)^w$ calculates the number of execution for all transactions triggered by a critical event type.

The second term $\left(\left(\sum_{k=0}^{q}(p+1)^k\right)-\left(\sum_{k=0}^{q}p^k\right)+q+1\right)^{t-w}$ takes into account all execution states for transactions triggered by a non critical event type leaving out states where the number of preemptions is between one and the allowed maximum.

The difference between PSTAT and PSTAT´´ is shown in table 4.5. The number of execution states in PSTAT´´ is considerably less than for the original PSTAT. In appendix B a comparison and verification of PSTAT´ and PSTAT´´ is done.

$$PSTAT''(p,q,t,w)=\left(\sum_{k=1}^{q}(p+1)^k\right)^{w}*\left(\left(\sum_{k=0}^{q}(p+1)^k\right)-\left(\sum_{k=0}^{q}p^k\right)\right)^{t-w}=$$

$$\left(\frac{(p+1)^{q+1}-1}{p}\right)^{w}*\left(\left(\frac{(p+1)^{q+1}-1}{p}\right)-\left(\frac{p^{q+1}-1}{p-1}\right)+2\right)^{t-w}\quad when\ \ p\geq 2 \tag{13}$$

$$PSTAT''(p,q,t)=\left(\frac{(p+1)^{q+1}-1}{p}\right)^{t}\ when\quad p=1$$

PSTAT´	PSTAT´´	p	q	t	w	%
371293	123201	2	2	5	2	66.8
102400000	3902240	2	3	5	2	61.9
25937424601	12552527375	2	4	5	2	51.6
371293	177957	2	2	5	3	52.0
102400000	5382000	2	3	5	3	47.4
259374246021	15988338025	2	4	5	3	38.5
37129	257049	2	2	5	4	30.7
1024000000	74240000	2	3	5	4	27.5
2593742601	20364093695	2	4	5	4	21.5

Table 4.5. Difference between PSTAT´ and PSTAT´´.

Chapter 5

Applicability of Constraints

This chapter investigates application of constraints aimed at improving the testability of real-time applications using an active real-time database system. Section 5.1 discusses effects of using constraints to improve testability of real-time applications. Section 5.2 presents guidelines for selecting constraint values. Finally, section 5.3 discusses ways to enforce the constraints.

5.1 Effects of Applying Constraints to Improve Testability

This section discusses effects that the constraints discussed in chapter four have on real-time applications, and on the underlying active real-time database system support. In section 5.1.1 several properties of real-time systems are discussed and selected for investigation. Selection of important systems service that need investigation is also covered. Section 5.1.2 discusses effects on the selected properties. Finally, section 5.1.3 discusses effects on scheduling.

5.1.1 Real-Time Properties and System Services to Investigate

A number of important properties that a real-time system must possess can be identified, e.g., predictability, efficiency, extensibility, dependability, responsiveness, and flexibility. Two properties have been selected for investigation: predictability and efficiency. The selection of predictability and efficiency is motivated by their role in achieving timeliness, i.e., the ability to complete tasks before their deadlines. A general definition of predictability is given by Stankovic [Sta90]:

> "It [predictability] means that it should be possible to show, demonstrate, or prove that requirements are met subject to any assumptions made."

Stankovic also points out that if only timing requirements are considered predictability can be defined as the ability to determine a task's completion time with certainty. Testing timeliness in a real-time system is the main concern in this work and, therefore, the latter definition of predictability is more appropriate for the purpose of this work.

A real-time system needs to be sufficiently efficient. Efficiency is measured by the systems ability to use its resources in an optimal way.

An active real-time database system (ARTDB) requires event-monitoring, scheduling, and rule-management. Of these services scheduling is the most interesting one to investigate. Scheduling can be sensitive to any constraints placed on system behavior and, therefore, effects that the constraints (see section 4.1) have on scheduling are investigated. Moreover, as discussed in section 3.1 testing of timeliness is the primary concern in this work and scheduling must guarantee timeliness.

5.1.2 Effects of Constraints on Predictability and Efficiency

This section discusses the effects that constraints to improve testability discussed in chapter four have on the properties selected section in 5.1.1.

Effects on Predictability

As discussed in section 2.1, correctness of a real-time system depends on its temporal correctness as well as its logical correctness. Therefore, it is important that a transaction in a real-time system completes before its deadline. In order to tell if a transaction completes before its deadline a predictable system is required. That is, based on assumptions about the behavior of the system it must be possible to predict with certainty the completion time of a transaction.

Constraining Preemptions

The presence of a constraint on the number of times a transaction can be preempted makes maintaining predictability difficult. The transaction model presented in section 3.3.2 assumes amongst other things that:

T1) Each transaction has a hard or soft deadline.

T2) Each transaction can be arbitrarily preempted.

T5) Worst-case execution time (wcet), deadline, and earliest start time are known for each transaction with a hard deadline.

A direct consequence of constraining the number of times a transaction can be preempted is that T2 cannot hold at all times. When a transaction has been preempted maximum number of times it must run to completion. The transaction model assumes that only worst-case execution time for hard transactions is known a priori. Therefore, the delay that a soft transaction can cause to a hard transaction is unbounded. Consequently no guarantee can be given on the completion time of hard transactions, and the system is no longer predictable. To achieve predictability under these conditions there must be a way to abort soft transactions in a bounded time.

Because worst-case execution time of all hard transactions is known the execution of hard transactions is still predictable when a maximum on the number of times a transaction can be preempted is enforced. If a hard transaction T_1, that has been preempted maximum number of times, is executing when a higher priority hard transaction T_2 arrives the time that T_1 can delay T_2 is known. If no transaction with higher priority than T2 arrives T2 will in the worst-case execute after the deadline of T1. Further, priority ceiling inheritance protocol [SRL90] (see section 2.1.2) or a similar protocol must be employed to maintain predictability in a system with shared resources. If the worst-case execution time for soft transactions is known the above discussion would apply to all transactions in the system.

The discussion in this section covers two scenarios. In the first scenario a soft transaction is running when a hard transaction arrives. The second scenario has a hard transaction running when a higher priority hard transaction arrives. Two additional scenarios are possible. A soft transaction is executing and another soft transaction arrives or a hard transaction executes and a soft transaction arrives. Neither of these scenarios are important in this context. A scenario with two soft transactions creates no problems because, as discussed in chapter two, a soft transaction may occasionally miss its deadline. The scenario where a hard transaction is executing and a soft transaction arrives is not problematic either because a soft transaction can never preempt a hard one.

Executing/Arriving	Soft	Hard
Soft	Not problematic because deadline can occasionally be missed.	Not problematic because a soft transaction cannot preempt a hard one.
Hard	Soft transaction can delay a hard for an unbounded time unless time to abort is bounded.	Because wcet is known predictability is maintained under the constraint.

Table 5.1 Summary of scenarios.

Constraining Transaction Instances

In an unconstrained system the scheduler tries to create a schedule from all admitted transactions. If earliest deadline first algorithm is used this means that, at each scheduling point, the scheduler selects the transaction with the earliest deadline from the list of admitted transactions. Placing no restrictions on which transactions are admitted to the scheduler creates a problem for maintaining predictability. Once the maximum number of concurrently executing transaction instances is reached no more instances of that type can execute. This means that a transaction instance cannot execute until another instance of the same type terminates. In the worst case a transaction must wait for the duration of all concurrently executing transactions. For soft transactions this does not cause a problem because they can occasionally miss their deadlines. The situation is more complicated for hard transactions. The impact of missing a hard deadline may be

catastrophic and, therefore, it must be guaranteed that these transactions meet their deadlines. In the transaction model discussed in section 3.3.2 it is assumed that:

T5) Worst-case execution times for all hard transactions are known.

T7) A transaction instance is triggered by a single event type which is either critical or non-critical.

This means that it is possible to known when transaction instances of the same type complete. To illustrate this consider a system with two transaction instances T_1, T_2, which are triggered by the same event type, and any number of transaction instance triggered by other event types. T_1 has a hard deadline and as a consequence of (2) T_2 must also have a hard deadline. T_1 is executing when T_2 arrives and only one instance is allowed to execute at any time. Since T_1 has a hard deadline the completion time of T_1 is known. Hence, it is possible to predict the completion of T_2. In the worst-case T_2 completes after all currently executing transactions[8].

If (2) does not hold, i.e., transaction instances triggered by the same type can be of different criticality, T_1 in the above example could have a soft deadline which means that its completion time is not predictable. This means that T_1 must be aborted in favor of T_2, and the time to abort T_2 must be bounded.

Constraining Observation Points

Constraining observation points means that asynchronous interrupts are delayed between observation points. Transactions that arrive are only detected at observation points. The length between each such observation point is therefore important. As long as this length is sufficiently short (see section 5.2) there are no problems.

[8] Assuming that the priority of T1 is lower than the priority of all other executing instances.

Effects on Efficiency

Efficiency in a non real-time transaction system can be measured in the number of committed transactions over a period of time. A real-time transaction system must meet deadlines and, therefore, it is of no use to achieve good throughput if deadlines are missed. An efficient real-time transaction system uses and allocates its resources in an optimal way.

Constraining preemptions

Constraining the number of times a transaction can be preempted can result in a less efficient systems. The decrease in efficiency occurs as the number of aborted soft transaction increases. As discussed in the previous section aborting soft transactions is needed to maintain predictability when preemptions and concurrent instances are constrained. As a result of assumption 4 in the execution model (see section 3.3.2) an aborted soft transaction is resubmitted. Under these circumstances system resources are not used in an optimal way because considerable effort is wasted on aborting and restarting transactions.

Constraining Instances

If predictability, in the presence of a constraint on the number of concurrently executing transaction instances triggered by the same type, is maintained by aborting soft transactions efficiency is effected in the same way as when preemptions are constrained. Otherwise, efficiency is no effected.

5.1.3 Effects of Constraints on Scheduling

A transaction can move through several states during its execution. In order to be executed a transaction has to be admitted to the scheduler where it is scheduled to start execution at a specific point in time. From the time a transaction starts to execute four

things can occur: the transaction executes until it has completed, the transaction is blocked when trying to access a resource that is locked by another transaction, the transaction is preempted and removed from the processor in favor of another transaction, or the transaction is aborted. Preemptive scheduling is in general responsible for deciding if a transaction completes, is preempted during execution, or aborted[9]. If a constraint on the number of time a task can be preempted is enforced preemptive scheduling no longer has total control over when transactions execute. Most of the time the scheduler can start execution of a new transaction, but if the running transaction has been preempted maximum times it must run to completion. During this period scheduling in non-preemptive (see section 2.1.2). Constraining the number of allowed concurrently executing transaction instances triggered by the same event type has a similar effect. Once the maximum is reached the scheduler cannot start another instance of the type that has reached the maximum. As discussed in section 2.1.2, the objective of a scheduler in a real-time system is to guarantee that transactions meet their deadlines. Therefore, of primary interest, with respect to effects of the constraints, are ways to ensure timeliness of transactions. The scheduler must guarantee all hard transaction in spite of constraints on allowed preemptions and concurrently execution transaction instances.

From the above discussion it is clear that under the constraints that improve testability scheduling is preemptive or non preemptive depending on the state of the system. Traditional scheduling algorithms assume that preemption is either allowed or not. Dynamic scheduling algorithms that execute transactions according to some priority use preemption to ensure that a higher priority transaction is serviced as soon as possible. The fact that scheduling is non-preemptive when the running transaction has been preempted maximum allowed number of times must be taken into account. Otherwise, transaction may miss its deadline if it executes after the running transaction.

Consider again the four scenarios presented in section 5.1.2. In the first scenario a soft transaction is running when a hard transaction arrives. If the soft transaction has not been preempted maximum number of times it is preempted and the hard transaction executes. The execution model (see section 3.3.2) assumes that if the soft transaction is

[9] The scheduler does not have complete control over this since the application programmer can, if desired, also control transaction execution.

preempted maximum number of times it is aborted. Moreover, the abort time is bounded. Therefore, it can be guaranteed off-line that no hard transaction will miss its deadline because of a maximum preempted soft transaction. The necessary condition is that no hard transaction has a tighter deadline than the time it takes to abort a soft transaction.

The second scenario has a hard transaction executing when a more urgent hard transaction arrives. If the running transaction has not been preempted maximum number of times this does not cause a problem because preference can be given to the arriving transaction. However, if the running transaction has been preempted maximum number of times the arriving transaction must wait until it completes. More precisely, if preemptions are allowed at arbitrary points a transaction can reach maximum number of preemptions at any time during its execution. This causes a problem if the arriving transaction cannot meet its deadline when started after the running transaction has completed. Two possible solutions to guarantee timeliness of the arriving transaction are suggested: i) preemption points are used to avoid the problem, and ii) imprecise computation techniques are employed (see section 2.1.3). A preemption point marks a point in the execution of a transaction where it can be preempted. Preemption points can either be inserted at arbitrary points or at points that split the transaction into equal duration intervals. For these intervals, if the worst-case execution time for the transaction is known, the worst-case execution time for each interval is also know. Further, a minimum execution based on the maximum number of allowed preemptions can be guaranteed. A transaction can always execute at least as many intervals as there are allowed preemptions. This avoids the scenario where a transaction is preempted maximum times during the beginning of its execution. Moreover, if the number of preemption points is equal to the number of allowed preemptions a transaction preempted maximum number of times is always executing in its final interval.

The second approach to this problem is to use imprecise computation techniques (see section 2.6). If a transaction, that has been preempted maximum number of times, can be prematurely terminated but still return a sufficiently precise result the problem is solved. The arriving transaction can immediately execute in preference to the running transaction despite the running transaction being preempted maximum number of times. However, this requires the ability to return an imprecise result at any state in a

transactions execution. As discussed in section 2.6, this requires monotone transactions, but these are rare. Another approach mention in section 2.6 is to skip certain computation steps, i.e., once a certain amount of computation is completed the transaction can return an imprecise result. If it can be guaranteed that an arriving transaction will meet its deadline after the running transaction finishes its mandatory computation part then this approach can be used. One possible solution is to disallow preemptions for the duration of the mandatory computation. During off-line analysis it is asserted that no transaction can miss its deadline because of the longest mandatory part. The third approach from section 2.6 is to use multiple versions of transactions. A primary transactions is normally executed but can be substituted for an alternate transactions. If the deadline of the primary transaction cannot be met the scheduler must choose to execute the alternate transaction instead.

The remaining two scenarios where a soft transaction is executing when another soft transaction arrives, and where hard transaction is executing when a soft transaction arrives provide no problems for the scheduler. The arriving soft transaction may miss its deadline, and it cannot preempt a hard transaction.

The discussion in the section can be summarized in the following points:

- Scheduling needs to take into account that it is not always possible to preempt the running transaction.
    - Using preemption points can ensure that deadlines are not missed because a transaction has been preempted maximum number of times.
    - Imprecise computation techniques can be used by the scheduler to ensure timeliness of hard transactions.
- When scheduling hard and soft transactions the time to abort soft transactions must be considered. The tightness of hard deadlines must never less than the upper bound for aborting a soft transaction. Alternatively, knowledge about worst-case execution time for soft transactions must be known.

5.2 Guidelines for Constraint Value Selection

When designing applications that use constraints to improve testability the feasibility of the design must be shown. Design guidelines with respect to the constraints aim at helping the designer to avoid constraint values that result in an application that is not feasible. An application that uses constraints in such a way that it is impossible to guarantee hard deadlines is not feasible. Necessary and sufficient conditions are given for selection of constraint values when all transactions have hard deadlines, periodic or sporadic arrivals, and deadlines are within respective periods. If soft transactions are included or deadlines are can be longer than periods only necessary conditions on constraint values are given. Applications that use constraint values that violate the necessary conditions cannot guarantee that all hard deadline are met.

Four different situations will be discussed and conditions on constraint value selection for each situation presented. Each situation is a selected combination of three transaction attributes. These transaction attributes are:

1) Hard or mixed (hard and soft) deadlines.

2) Periodic or sporadic arrivals.

3) Deadlines for periodic transactions are either i) less than or equal to period, or ii) greater than period. A minimum interarrival time is assumed to be known for sporadic transactions (see assumption T5 in section 3.3.2). Therefore, their deadlines can be i) less than or equal to minimum interarrival time, or ii) greater than minimum interarrival time.

Real-Time systems that only have transactions with soft deadlines are not considered in this dissertation. Therefore, with respect to constraint value selection, is it sufficient to consider systems with only hard deadlines or with mixed deadlines. Similarly, only periodic and sporadic arrivals are assumed (see assumption T4 in section 3.3.2) and therefore aperiodic arrivals are not included in (2).

In accordance with the discussion in section 5.1.3 it is assumed that preemption points are used to control when transactions may be preempted. This is necessary because otherwise a transaction can reach maximum preemptions at an arbitrary point in its execution. When that occurs scheduling in non-preemptive.

Situations are presented in order of simplicity. The simplest situation occurs when we have transaction with hard deadlines and periodic arrivals where deadlines are equal to or less than the period. Extending this situation to also allow sporadic arrivals results in situation two. The third situation is concerned with periodic and sporadic arrivals where deadlines may be greater than period/minimum interarrival rate. Finally situation four covers the most difficult situation where the transaction set has mixed deadlines. In situation one and two necessary and sufficient conditions for constraint value selection are presented. Only necessary conditions are presented for situations three and four.

Situation 1: Hard deadlines with periodic arrivals where deadlines are less than or equal to the period.

Observation interval

To guarantee that observation interval length does not cause a constrained system to miss deadlines that an unconstrained system would not the constrained system must observe precisely enough to detect events as they occur. Therefore, a necessary and sufficient condition for observation interval length in a system with a set S of transactions is:

$I_o = gcd^{10}(period(T_1),...,period(T_n))$ where $T_1...,T_n$ are transaction in S.

The reason for this is that observation points will match request points for all the transactions. The necessary requirement for this to hold is that there is no jitter in the arrivals, i.e., there is no variance in request times.

[10] $gcd(x_1...x_n)$ is the greatest common divisor of $x_1..x_n$.

Preemption interval

Given a set S of transactions and an observation interval I_o a necessary condition on selection of preemption interval length I_p is:

$$\forall t : t \in S \ \ (I_p \leq deadline(t) - wcet(t))$$

In other word preemption interval length must not exceed the shortest slack in the set S. However, I_p is not a sufficient condition to guarantee that deadlines met by an unconstrained system will be met when preemptions are constrained. A sufficient condition must take the entire transaction load into account. According to Liu and Layland [LL73], if all transactions have periodic arrivals and deadlines are within respective periods, a sufficient schedulability test under the rate-monotonic approach is:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1)$$

In the above formula C_i is worst-case execution time of a transaction and P_i its period.
This has been extended by Sha et. al. [SRL90] to include the effect of blockings. Under the priority ceiling protocol a sufficient schedulability test is:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} + \max_{i=1}^{n}\left(\frac{B_i}{P_i}\right) \leq n\left(2^{\frac{1}{n}} - 1\right)$$

The difference from the original schedulability test is that a blocking term B_i is introduced. For each transaction B_i is the longest blocking that this transaction can be subject to.

The effect of using preemption intervals can be integrated into this schedulabilty test[11]. A lower priority transaction can only delay a higher priority transaction for the duration of a single preemption interval. Moreover, a transaction can only be delayed once during its entire execution by a lower priority transaction. Therefore, a preemption interval can be seen as a blocking where each transaction except the lowest priority one has a blocking term equal to the preemption interval length. A modified schedulabilty test where preemption intervals are included is:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} + \max_{i=1}^{n}\left(\frac{B_i}{P_i}, \frac{I_p}{P_i}\right) \leq n\left(2^{\frac{1}{n}} - 1\right)$$

Maximum preemptions

Based on I_p the number of allowed preemptions can be selected. Given a set S of transactions the maximum number of allowed preemptions M is set to:

$$M = \left\lceil \frac{\max(wcet(t_1),...,wcet(t_n))}{I_p} \right\rceil - 1 \ \textit{where}\ t_1..t_n \in S$$

This divides the longest transaction into intervals of length I_p. For this transaction to complete the number of allowed preemptions must be set to one less than the number of intervals.

Maximum concurrent instances

If all periodic transaction have deadlines that are less than or equal to their period only one instance can be active at any time. Therefore, it is sufficient to set the maximum number of concurrently executing instances triggered by the same type to one (q=1).

[11] Suggest by Jörgen Hanson through personal communication.

Situation 2: Hard transactions with periodic and/or sporadic arrivals where deadlines are equal to or less than period or minimum interarrival time.

Observation interval

A discussed in situation 1 a necessary and sufficient condition of observation interval length for periodic transactions is to chose the greatest common divisor of periods as the interval length. This does not hold when sporadic arrivals are allowed. The reason is that transactions with sporadic arrivals can be requested at any point in time as long as two requests are separated by a minimum interval. Choosing the greatest common divisor of periodic arrivals and sporadic minimum interarrival rates is not sufficient to guarantee detection of arrivals as they occur. Assume that observation interval length is chosen as:

I_o = gcd(period/minimum interarrival time(T_1),...,period/minimum interarrival time(T_n))
where T_1...,T_n are transaction in S

A sporadic arrival is in the worst-case detected I_o time units after its actual occurrence. To handle this potential delay deadlines of transactions with sporadic arrivals must be adjusted. Therefore, an additional condition is needed. Given a set S´ of transactions with sporadic arrivals:

$\forall t : t \in S'$ $(\text{deadline}(t) \leq I_p + \text{period}(t))$ where period is equal to minimum interarrival time.

However, deadlines are required to be within respective periods and, therefore, the additional delay caused by I_p must only be externally visible. When assessing schedulability all transactions are treated as if they were periodic with deadlines inside periods, but externally deadlines for sporadic transactions are expected to complete in the worst case at I_p + period(t).

Preemption interval

Selection of preemption interval length is done as in situation 1.

Maximum preemptions

Using I_p maximum preemptions are derived in the same way as in situation 1.

Maximum concurrent instances

Periods and minimum interarrival rates are within deadlines, therefore, the same selection is done as in situation 1.

Situation 3: Hard transactions with periodic and/or sporadic arrivals where deadlines can be greater than period/minimum interarrival time.

Selection of observation interval is done in the same way as in situation 1. The schedulability test used in situation 1 and 2 to determine preemption interval length and maximum number of allowed preemptions does not apply for this situation. The reason is that this schedulability test requires deadlines to be within periods. Therefore, only the necessary condition given in situation 1 for selection of preemption interval length holds for this situation.

$$\forall t : t \in S \quad (I_p \leq deadline(t) - wcet(t))$$

Using this necessary condition maximum allowed preemptions can be selected as in situation 1.

Maximum concurrent instances

If deadlines can be greater than period/minimum interarrival rate more than one transactions instance triggered by the same event type can execute concurrently. The reason is that requests for an instance can arrive before a currently executing instance of the same type completes. For a set S of transactions the maximum number of concurrently executing instances is determined by:

$$\left\lceil \max\left(\frac{deadline(t_i)}{period(t_i)},\ 0 < i \le n \right) \right\rceil \ where\ t_1..t_n \in S$$

For sporadic transactions period(t) is replaced by minimum interarrival time.

Situation 4: Hard and soft transactions with periodic/sporadic deadlines.

When the transaction set is a mixture of hard and soft transactions constraint values are selected according to situations 1 and 2 respectively. However, only properties of hard transactions are used to select preemption interval length, and consequently the number of allowed preemptions is only based on hard transactions. The reason is that soft transaction that reach maximum preemptions are always preempted if a hard transaction arrives. Therefore, they need not be considered in this context. Moreover, since RMA is not suited for a mixed transaction set sufficient conditions cannot be given for this situation. Therefore, selection of preemption interval length and maximum number of allowed preemptions is done as in situation 3.

Example

An example of how these guidelines are used is given below. The properties of th example system match those required for situation 1. Therefore, constraint value selection is done according to situation 1.

Transaction	Period	WCET	Deadline	Arrival	Type
T1	850 ms	200 ms	850 ms	Periodic	Hard
T2	850 ms	100 ms	850 ms	Periodic	Hard
T3	70 ms	5 ms	70 ms	Periodic	Hard
T4	160 ms	20 ms	160 ms	Periodic	Hard

Table 5.1. Example system

The transactions in table 5.1. fall under situation 1 and accordingly observation interval length is chosen as:

$$I_o = \text{gcd}(850,850,70,160) = 10\,\text{ms}$$

To select preemption interval length schedulability of the transaction set is first checked. All transaction are guaranteed to meet their deadlines if:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

$$\sum_{i=1}^{n} \frac{200}{850} + \frac{100}{840} + \frac{5}{70} + \frac{20}{160} \leq 4(2^{\frac{1}{4}} - 1)$$

$$0.55 \leq 0.76$$

This means that selection of preemption interval must satisfy:

$$0.55 + \max_{i=1}^{n}\left(\frac{I_p}{P_i}\right) \leq 0.76$$

Selecting I_p=12 ms gives:

$$0.55 + \frac{12}{70} \leq 0.76$$

$0.72 \leq 0.76$ which means that all deadlines are guaranteed to be met.

From this maximum number of allowed preemptions can be derived.

$$M = \left\lceil \frac{wcet(T_1)}{I_p} \right\rceil - 1 = \left\lceil \frac{200}{12} \right\rceil - 1 = 16$$

Finally, because all deadlines are within periods q is set to one.

5.3 Enforcement of Constraints to Improve Testability

This section discusses different methods that can be used for enforcing the constraints discussed in section 2.5.4 and chapter 4. The methods that are discussed are prevention, avoidance, and detection and recovery.

5.3.1 Applicability of Enforcement Methods

Enforcing the constraints can be done in three ways. Firstly, before run-time the system is conditioned in such a way that the constraints never can be violated. Secondly, during run-time mechanisms for avoiding constraint violations are employed. Finally, the system detects constraint violations during execution and perform a recovery action.

A violation on concurrently executing instances triggered by the same event type can be prevented by ensuring that no more than one instance executes at any time. To achieve this requires constraints on characteristics of transaction instances as well as on resource allocation. Violation of the maximum preemptions can also be prevented. This

requires each transaction to be split up into equally many intervals as there are allowed preemptions. Preemption is only allowed at points where two such intervals meet.

Avoiding that the constraints are violated requires that mechanisms are inserted into the system that can control the flow of events and the execution of transactions. The objective is to shield the system during times when more transaction instances are requested than can be serviced. Further, it must be guaranteed that a transaction is not preempted more than the maximum allowed times.

Normally the detection and recovery schema includes a recovery mechanism that can bring the system back to an operational state. For example when a deadlock is detected the system recovers by terminating some transactions. Recovery of this type is not applicable in this work. Once the constraints have been violated there is no return. The system must indicate to the user that the constraints have been violated, and that results from testing are no longer reliable.

5.3.2 Enforcement by Prevention

Concurrent instances triggered by the same type

A necessary condition for violation of maximum concurrently executing instances is that more than one instance triggered by the same type can execute concurrently. If this does not hold then the constraint can never be violated. This can be achieved by modifying the transaction and resource models from section 3.3.2. First, if only periodic transactions with deadlines within periods are allowed, and these are discarded if they miss their deadlines only one instance of each type executes at the same time. If sporadic transactions are allowed then the following properties must hold:

1) Transaction instances triggered by the same type have the same priority. This means that a transaction instance cannot be preempted in favor of another instance triggered by the same type.

2) All transaction instances triggered by the same event type have the same resource requirements.

3) All resources must be allocated to a transaction before it executes and it holds all its resource for the duration if its execution.

As long as these properties hold the concurrent execution of transaction instances triggered by the same type is not possible. A transaction that starts execution executes to completion unless it is preempted by another transaction. But that transaction can never be triggered by the same event type. Moreover, other instances triggered by the same type will never execute because they require the same resources as the running instance.

Maximum preemptions

Preemption points were discussed in section 5.1.3. If each transaction is divided into as many equal length intervals as there are allowed preemptions no transaction can be preempted more than the allowed maximum. Once a transaction has been preempted maximum times it is guaranteed to be executing in its last interval. Since no preemptions are allowed during each interval the transaction finishes its execution uninterrupted. The length of preemption intervals is covered in section 5.2.

5.3.3 Enforcement by Avoidance

In the previous section ways to prevent the violation of the constraints were discussed. In this section ways to avoid constraint violation are discussed. Two mechanisms: adaptive filters (see section 2.1.4) and admission control (see section 2.2.4) are suggested for handling violations of maximum number of concurrently executing instances.

Maximum Concurrent Transactions

Enforcing the number of concurrently executing transaction instances triggered by the same event type requires employing run-time mechanisms that ensure that the number of executing instances does not exceed the allowed maximum. This can be done by: i) filtering events to ensure that the number of events presented to the system matches the number of allowed transaction instances; or ii) filtering transactions before they are admitted to the scheduler. Adaptive flow control filters discussed in section 2.1.4 are well suited to filter out surplus events. For filtering transaction instances admission control mechanisms (see section 2.2.4) are a suitable choice. Both approaches must use some policy for filtering events or transactions. A policy for an adaptive filters determines which event instances are dismissed. Similarly, a policy for an admission controller determines which transaction instances are admitted to the scheduler. The objective of these mechanisms is to ensure that no more than the allowed number of instances execute. At the same time no hard deadlines must be missed because of this.

Policy for admission control

If the number of submitted transaction instances triggered by one event type exceed the allowed maximum surplus instances must be rejected. The selection of instances to reject depends on semantics of the triggering event. Semantics of certain event types allow rejection of surplus instances, e.g., event types that trigger idempotent actions. For example, two consecutive events that trigger an action to open a valve can be replaced by a single action. Another example of event types that allow surplus transaction instances to be rejected are event that trigger actions such as reading of temperature or movement monitoring. For these event types the newest instance can replace older ones as long as deadlines are met.

The simplest policy is to reject transaction instances according to semantics of the triggering event type. A more sophisticated approach is to also use knowledge about the state of the system. For example, instead of rejecting all surplus transaction instances

execution of an instance can be delayed until another instance of the same type completes or instance that are already admitted can be replaced by a newer instance.

From the above discussion policies for rejecting transaction instances are suggested:

1) For transaction instances triggered by idempotent event types the newest instance is admitted to the scheduler and all other instances are rejected.

2) For transaction instances triggered by event types where order of events is not important admitted instances are selected according to their deadline. Instance with earlier deadlines are favored. The maximum number of instances that can be admitted is: maximum allowed - (executing + admitted).

3) For transaction instances triggered by an event type that requires all requests to be serviced two possibilities exist: i) if the event type is noncritical all surplus instances are rejected and eventually resubmitted, and ii) if the event type is critical it cannot be rejected. For option (i) the number of instances that can be admitted is: maximum allowed - (executing + admitted).

4) If a deadline of a critical instance is such that it can meet it if executed after a currently executing instance of the same type finishes then it does not have to be rejected. The necessary condition is: deadline(new instance) >= deadline(old instance) + wcet(new instance), and earliest release time is less than or equal to deadline(old).

5) An extension to (4) is to remove transaction instances from the list of admitted transactions. When a new instance arrives which cannot be admitted then an existing instance of the same type is removed from the list of admitted transactions. The has no effect on other admitted transactions if the new transactions has the same requirements (wcet, deadline, and resource requirements) as removed one.

Policy for adaptive filter

An adaptive filter discards event instances is a similar way as transaction instances are rejected by an admission controller. The selection of instance to discard is based on event type semantics. Policies (1) and (2) used for admission control also apply to an adaptive filter.

Maximum Preemptions

Enforcing a maximum on the number of times a transaction can be preempted requires a check to be done on how often the running transaction has been preempted. This must be done before the scheduler informs the dispatcher to preempt the running transaction. For this purpose the number of times a transaction has been preempted must to be stored. The control block of a transaction is the most appropriate place to store this information. After each observation point the scheduler selects the next task to run. If the running transaction has been preempted maximum times then the scheduler must decide if the actions described in section 5.1.2 are to be taken or if the transaction is allowed to complete. If there is a need to either abort a soft transaction or to apply imprecise computation techniques the scheduler must take necessary actions. Once the processor has been released the scheduler informs the dispatcher to execute the transaction selected to run next.

5.3.3 Enforcement by Detection and Recovery

A computationally cheaper method is to detect the violation of the constraints and perform some recovery. As described in section 5.3.1 the only recovery action done in this case is to notify the application that the constraints have been violated.

Max Instances

To enforce the max instances constraint a record of how many instances of each type are currently executing is needed. This requires knowledge about which event type triggered a given transaction. Given that this information is available, the dispatcher records each time a transaction terminates as well as each time a transaction is allocated to the processor. A watchdog process is invoked before a transaction is allocated to the processor. The watchdog process checks to see if to many instances of some type are concurrently executing. If this is the case the application is notified.

Max Preemptions

The scheduler is well suited to detect a violation of the preemption constraint. Before sending a request to the dispatcher to preempt the running task the scheduler checks the control block of the transaction. If the transaction has already been preempted maximum number of times the application is notified that the constraint has been violated. It is then the applications task to notify the tester. Alternatively the dispatcher could check how often a transaction has been preempted. The dispatcher receives a request from the scheduler to preempt the running transaction. It performs the context switch and then checks the control block of the transaction to see if it has been preempted more than maximum times. As before the application is notified.

Chapter 6

Results

This chapter presents a summary of the work performed and highlights the results. Further, a partial validation of the results is performed.

6.1 Validation and Refinement of Test Effort Upper Bound

Validation of existing formulas (see section 4.1) for the test effort upper bound resulted in discovery of two errors. First, formula for potential preemptions gives fewer preemptions than there actually are. Second, event types are not separated from event instances. Corrected formulas were produced (see section 4.2).

The number of potential blockings in the orginal formula only covers the case where a single resource is locked. A new formula that calculates potential blockings when any number of resources are locked was produced (see section 4.3).

As suggested by Mellin [Mel96], the test effort can be reduced by only testing hard transactions completely. One way of reducing the effort is to only include a subset of possible execution states. Timeliness of hard transactions is of utmost importance while soft transactions may occasionally miss their deadlines. Therefore, it is feasible to not fully test soft transactions. The approach taken is to leave out execution states where soft transactions are not preempted maximum number of times. Instead of testing all combinations of soft and hard transactions only combinations where the soft transaction is not preempted or preempted maximum number of times are included. The reason is that these are the only combinations that can effect the execution of hard transactions. The resulting formula for execution states has been produced (see section 4.3.2).

6.2 Investigation of Effects On System Properties and Services

The effects on predictability, efficiency, and scheduling have been investigated in this work. Further, ways to counter the effects have been discussed.

6.2.1 Predictability

Constraining then number of times a transaction can be preempted and the number of concurrently executing transaction instances triggered by the same event type has a negative effect on predictability. Two problems arise:

i) If the worst-case execution time for soft transactions is not known a soft transaction that has been preempted maximum number of times can delay execution of a hard transaction for an unbounded time.

ii) When the maximum number of transaction instances triggered by the same type are executing a new instances cannot execute until a currently executing instance completes.

Maintaining predictability under these circumstances requires that:

i) There exist a bound on the time to abort a soft transaction. If soft transaction is executing when a hard transaction arrives the soft transaction is aborted in a bounded time. erefore, it is possible to guarantee that no hard transaction misses its deadline because of a maximum preempted soft transaction.

ii) Worst case-execution time for all hard transactions is known and transactions instances triggered by the same type cannot have different criticality. This makes it possible to predict when a transaction instance, of a type that has reached the maximum, can complete.

Alternatively, if worst-case execution times are known for both hard and soft transactions the system is predictable under these constraints.

6.2.2 Efficiency

In this dissertation it is assumed that a soft transaction that is aborted by the system in favor of a hard transaction is resubmitted if needed. As discussed in the previous section a soft transaction that reaches maximum preemptions is always preempted if an hard transaction arrives. If soft transactions are frequently aborted and restarted system resources are not being used optimally and efficiency decreases.

6.2.3 Scheduling

Traditional scheduling algorithms assume that transactions are either preemptable or not. Defining a maximum on the number of times a transaction can be preempted introduces nonpreemptable periods. A transaction that has been preempted maximum number of times must run to completion. In a similar way defining a maximum on the number of concurrently executing transaction instances triggered by the same type introduces nonpreemptable periods. Transactions, triggered by an event type for which the maximum is reached, cannot be preempted in favor of another instance triggered by that same type. It must be guaranteed that no hard transactions miss their deadlines. This can be done by asserting certain properties off-line. As discussed in the previous section soft transaction are aborted in a bounded time. If no hard transactions has a tighter deadline than this bound then it is possible to guarantee that deadlines cannot be missed because of maximum preempted soft transactions. It must also be asserted that no hard transaction can miss its deadline because of another maximum preempted hard transactions. Two solutions are suggested: i) preemption points, and ii) imprecise computation techniques. If each transaction is split into as many event length intervals as there are allowed preemptions, and preemption is only allowed where two intervals meet then the following holds; A transaction that has been preempted maximum number of times is always executing in is final interval. This requires that no transaction has a

tighter deadline than the longest interval. Otherwise, a hard transactions can causing another hard transactions to miss its deadline (see section 5.2). The second solution is to use imprecise computation techniques to terminate execution with imprecise results. This makes it possible to execute hard transaction with a tight deadline in preference to a maximum preempted hard transaction.

6.3 Guidelines for constraint value selection

Guidelines for constraint value selection have been presented. The guidelines define necessary and sufficient conditions for selection of values if all transactions have hard deadlines, periodic or sporadic arrivals, and deadlines that are within their respective periods. For sporadic arrivals or a mixed transaction set with hard and soft transaction only necessary conditions are presented. The necessary and sufficient condition for value selection is based on selecting observation interval length as the greatest common divisor of periods. This makes it possible to observe events as they occur. Further, preemption intervals are incorporated into a schedulability test for the rate-monotonic algorithm. A set of transaction that are scheduled using the rate-monotonic algorithm is guaranteed to meet all deadlines if observation interval length is equal to the greatest common divisor of periods and the following holds:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} + \max\left(\frac{B_i}{P_i}, \frac{I_p}{P_i}\right) \leq n\left(2^{\frac{1}{n}} - 1\right)$$

In the above formula C_i is the worst case execution time; B_i is a blocking term; I_p is preemption interval length; and P_i is the period of a transactions.

Conditions of selection of constraints for situations where deadlines can be longer than periods or the system has a hard and soft transactions have also been covered. Only necessary conditions are given for these situations.

6.4 Enforcement of Constraints

Three possible methods for enforcing the constraints have been discussed. Prevention conditions the system before run-time in order to make sure that the constraints cannot be violated. Avoidance employs mechanisms at run-time that make sure that constraints are not violated. Finally, detection and recovery detects violations during execution and performs a recovery actions.

6.4.1 Prevention of Constraint Violations

Violation of maximum allowed concurrently execution transaction instances triggered by the same event type can be prevented in two ways:

1) If all transaction instances are periodic and deadlines are with respective periods then only one instance can execute at a time. This also requires that transactions that miss their deadlines be discarded.

2) Requiring all sporadic transaction instances triggered by the same type are to have the same priority. This means that an instance cannot be preempted in favor of another instance triggered by the same type. Hence, only one instance of each type can execute at any time. Further, each transaction instance triggered by the same event type must have the same resource requirements.

Preventing violation of the maximum number of times a transaction can be preempted is possible using preemption points. If each transaction instance is divided by preemption points into equally many intervals as there are allowed preemptions then the constraint can never be violated.

6.4.2 Avoidance of Constraint Violations

Two mechanisms, adaptive filters and admission control, were suggested for avoiding violation of concurrent transaction instances constraint. Violations are avoided through an admission controller by rejecting transaction instances. An instance that cannot be executed because the maximum number of concurrently executing instances for this type is reached is rejected. The selection of which instance to reject is done according to a policy. Policies for rejecting transaction according to event type semantics have been suggested. Further, policies that use information about the state of the system to decide if an instance should be rejected are presented. (see section 5.3.3). An adaptive filter controls the number of event instances to prevent more transaction instances than allowed to be admitted to the scheduler. This is done by discarding instances according to a defined policy.

6.4.3 Detection and Recovery of Constraint Violations

The scheduler was identified as the most appropriate mechanism to avoid violation of the maximum preemption constraint. As for detection and recovery it was noted that recovery in the traditional sense is not useful. Once a constraint violation has been detected the application is notified and testing is terminated. The detection of constraint violation is left to the scheduler and a special watchdog process. The scheduler detects violation of the maximum preemptions constraint while the watchdog process is responsible for detecting violations to maximum concurrent instances.

6.5 Validation of Results

This section presents a partial validation of results from this work. The validation is done by performing a case study of a simple event-triggered real-time control application. Throughout this section the rate-monotonic algorithm (RMA) [LL73] is used to schedule transactions. Since RMA is not suitable for a mixed load of hard and soft transctions only those results that are not related to soft transactions can be validated. The following results are validated:

- Upper bound on test effort for event-triggered systems with only hard transactions.

- The need for preemption points.

- Guidelines for constraint value selection.

Process Control System

Through the rest of this section a simple real-time control applications is used to validate the results presented in previous sections of this chapter. The control application controls a conveyor belt that carries objects over a weight sensor and redirects them to other conveyor belts. Objects are placed on the belt by a human operator into placeholders. The length between placeholders is constant. Normally the operator fills each placeholder with an object. The objects pass over a weight sensor and are redirected to other belts according to their weight. The number of objects that are allowed to enter other conveyor belts is restricted. An object that is routed to an overloaded belt is redirected back to the human operator. The control system has five transactions. Transaction T_1 processes input from the weight and stores the weight value into a database object. Transaction T_2 reads this value and selects a conveyor belt to direct the object to. Transaction T_3 is executed if the human operator presses an emergency stop button to halt the conveyor belt he is placing objects on. Transaction T_4 is triggered by a global emergency stop that halts all conveyer belts. Finally, transaction T_5 controls the admission of objects onto conveyor belts. All transaction have a hard deadline. All transactions are treated as if they had periodic arrivals. Transaction T_1 and T_2 share a database object and require exclusive access to it. Since T_1 and T_2 have the same period and T_2 has a higher priority than T_1 no blocking can occur between T_2 and T_1. Therefore, no blocking term is included in the schedulability tests. In this example a sporadic server is assumed to be used.

Transaction	Period	WCET	Deadline	Criticality	Type	Priority
T_1 (Read weight)	960 ms	170 ms	960 ms	Hard	Periodic	5
T_2 (Redirect)	960 ms	210 ms	960 ms	Hard	Periodic	4
T_3 (Local stop)	40 ms	3 ms	40 ms	Hard	Sporadic	2
T_4 (Global stop)	30 ms	2 ms	30 ms	Hard	Sporadic	1
T_5 (Overflow)	480 ms	10 ms	480 ms	Hard	Periodic	3

Table 6.1. Simple Process Control System

Constraint value selection

In the following sections it is assumed that constraint values have been selected according to situation 2 in section 5.2. This given the following values:

I_o = gcd(period(T_1), period(T_2), period(T_3), period(T_4), period(T_5))=
gcd(960, 960, 40, 30, 480) = 10 ms

To select preemption interval length schedulabilty of transactions is table 6.1 is first examined. According to Liu and Layland [LL73] all transaction will meet their deadlines if[12]:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1)$$

$$\sum_{i=1}^{n} \frac{170}{960} + \frac{210}{960} + \frac{3}{40} + \frac{2}{30} + \frac{10}{480} \leq 5(2^{\frac{1}{5}} - 1)$$

$0.56 \leq 0.74$ which means that all transactions are guaranteed to meet their deadlines.

According to guidelines in section 5.2 selection of preemption interval must satisfy:

[12] In the work by Liu and Layland T_i is used to indicate transaction periods. Since T_x is used to identify transaction is this work P_i is used to indicate transaction periods.

$$0.56 + \max_{i=1}^{n}(\frac{I_p}{P_i}) \leq 0.74$$

Selecting I_p = 5 ms gives:

$$0.56 + \frac{5}{30} \leq 0.74$$

$0.73 \leq 0.74$ which means that all deadlines are guaranteed to be met.
According to the guidelines in section 5.2 maximum allowed preemptions are derived from I_p using the longest worst-case execution time in the transaction set.

$$p = \left\lceil \frac{wcet(T_2)}{I_p} \right\rceil - 1 = \left\lceil \frac{210}{5} \right\rceil - 1 = 41$$

The number of concurrently executing transactions instances triggered by the same type (q) is set to one since all deadlines are within periods.

6.5.1 Test Effort Upper Bound

This section validates the test effort upper bound presented in section 4.2. This is done using the control application presented above. As discussed in section 2.4, assuming that no constraints for improving testability are imposed on this system a lower bound for the required test effort is defined by Schütz [Sch93] to be $(s+1)^n$ where s is the number of observation intervals and n the number of independent events. If we define an action interval g_a to be equal to the longest period of table 6.1 and observe with precision I_o then according to Schütz $s = \frac{g_a}{I_o} = \frac{960}{10} = 96$. This gives a lower bound on the test effort:

$(s+1)^n = (96+1)^5 = 8.587.340.257$ test cases.

The actual number of test cases required is much larger because of the number of possible synchronisation sequences. However, by using the method described in this work an upper bound on the test effort for our control application can be calculated using FSTAT''' (see section 4.3.1). The values of n and s are the same as in the lower bound calculation. For p and q the values selected above are used. Since there is only one shared resource (data object shared between T_1 and T_2) C containts {2,3,4,5} and O contains {1,1,1,1}.

$$FSTAT'''(n,s,p,q,t,C,O) = 97^5 * 26 * (\frac{11^2 - 1}{10})^5 = 5.5556 * 10^{16}$$

As already pointed out in section 4.1.1 FSTAT''' is conservative since not all combinations of blockings and preemptions are possible. Also, by taking into account the specifics of the control applications we can see that the actual number of test cases needed to much smaller than calculated by FSTAT'''. For example, when discussing properties of the control applications it was determined that no blocking can occur, but according to FSTAT''' even transaction that do not share a resource can block each other. A more detailed examination of the upper bound can be performed by using a smaller example. Therefore, a subset of the transactions in table 6.1 are selected for investigation.

Transaction	Period	WCET	Deadline	Criticality	Type	Priority
T_3	40 ms	3 ms	40 ms	Hard	Sporadic	2
T_4	30 ms	2 ms	2 ms	Hard	Sporadic	1

Table 6.2. subset of process control application

Selecting constraints values as before gives:

I_o = gcd(period(T_3), period(T_4))=gcd(40, 30) = 10 ms

The maximum preemtion interval length is determined as before by:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

$$\sum_{i=1}^{n} \frac{3}{40} + \frac{2}{30} \leq 2(2^{\frac{1}{2}} - 1)$$

$0.14 \leq 0.828$ which means that all transactions are guaranteed to meet their deadlines. This means that selection of preemption interval must satisfy:

$$0.14 + \max_{i=1}^{n}(\frac{I_p}{P_i}) \leq 0.828$$

Selecting I_p = 20 ms gives:

$$0.56 + \frac{20}{30} \leq 0.828$$

$0.81 \leq 0.828$ which means that all deadlines are guaranteed to be met.

The above situation could be scheduled without preemptions, but for the purpose of this example a smaller I_p of 2 ms is selected[13].

$$p = \left\lceil \frac{wcet(T_3)}{I_p} \right\rceil - 1 = \left\lceil \frac{3}{2} \right\rceil - 1 = 1$$

As before q=1 and t=5

Selecting g_a as 30 ms we get $s = \frac{g_a}{I_o} = \frac{30}{6} = 5$

T_3 and T_4 share no resources and therefore no blocking can occur.

[13] Any interval length smaller than 22 ms is valid.

Using FSTAT''' this gives an upper bound of:

$$FSTAT'''(n,s,p,q,t,C,O) = 6^2 * (\frac{2^2 - 1}{1})^2 = 36 * 9 = 324 \text{ test cases.}$$

Each test case is shown in table 1 in appendix E. Further, unnecessary test cases are discussed. This gives a comparison of the upper bound and the actual number of test cases that are needed.

6.5.2 Preemption Points

In this section the validity of preemption points is discussed. It is shown that preemption points are necessary to guarantee hard deadlines in a system that uses constraints to improve testability. Further, schedulability of a system using preemption points is investigated.

At the beginning of section 6.5 it was shown that the process control application passes the sufficient schedulability test for RMA. This means that all deadlines are guaranteed to be met. In section 6.5.1 observation interval length was chosen as the greatest common divisor of transaction periods. This means that transactions are observed as they occur and all deadlines met by an unconstrained system are also met when observations are constrained. In section 5.3 it was argued that defining preemptions points is necessary when the number of allowed preemptions is constrained. Otherwise, a transaction can reach maximum preemptions at an arbitrary point in its execution. This can be illustrated using the simple control application example. Assuming that preemptions are allowed at arbitrary points maximum number of allowed preemptions cannot be selected using the guidelines from section 5.1. During the period of transaction T_2 transaction T_3 can be requested

$$\frac{period(T2)}{period(T3)} = \frac{960}{40} = 24 \text{ times.}$$

Similarly, transaction T_4 can be requested

$$\frac{period(T_2)}{period(T_4)} = \frac{960}{30} = 32 \text{ times.}$$

Each of these requests can potentially cause T_2 to be preempted. As an example assume that maximum number of allowed preemptions is set to four. In table 6.2 execution of transactions T_2, T_3, and T_4 is shown. The time intervals where each transaction execute is indicated by x->y meaning that the transaction executed from time x to time y. A missed deadline is indicated by --time--. As table 6.2 shows this results in T_2 reaching maximum preemptions after 80 ms and cause T_3 and T_4 to miss several deadlines.

Transactions								
T_2	5->30	32->40	43->60	62->80	83->225			
T_3	2->5	40->43	80->83	--120--	--160--	--200--		
T_4	0->2	30->32	60->62	--90--	--120--	--150--	--180--	--210--

Table 6.3. Execution trace of simplified process control application

The situation presented in table 6.3 is an extreme one where allowed preemptions are selected without any consideration to the transaction load. However, in general it is difficult to select a correct number of preemptions. This is especially true if sporadic arrivals are allowed, since the precise points in time when preemption could potentially occur are not available a priori.

It has been argued in section 5.2 that to solve this problem the points in time where preemption is allowed must be constrained. Moreover, a necessary and sufficient condition for selection of preemption intervals from which the number of allowed preemptions can be chosen has been presented. This can be validated with respect to the control application example. In appendix F a timeline is shown for transactions in the control application. In the presence of the constraints all transactions meet their deadlines.

6.6 Related Work

Related work can be divided into two categories: i) work that explicitly addresses reduction of test effort for real-time systems by constraining system behavior, and ii) work that reduces test effort by testing a sub-set of possible behaviors.

6.6.1 Reducing Test Effort by Using Constraints

The only known work that deals with reduction of test effort for event-triggered real-time systems is done by Mellin [Mel96]. In his work Mellin discusses how an active real-time database system can be used to support constraints that reduce the test effort. By applying constraints to system behavior an upper bound on the test effort for event-triggered real-time systems is formulated. In contrast to Mellin's work this work focuses on the applicability of constraints to improve testability. This work explicitly defines a model for applications that use constraints to improve testability. Moreover, investigation of effects that the constraints have on predictability, efficiency, and scheduling is done in this work. This work increases applicability of using the constraints to improve testability by allowing more than one resource to be locked at the same time.

In work by Schütz [Sch95], testability of time-triggered real-time systems is compared to an unconstrained event-triggered systems. The test effort for time-triggered systems is expressed as being proportional to the number of observable environmental states. Further, a lower bound on the test effort for event-triggered systems is given. In contrast to the work done by Schütz [Sch95] this work addresses how testability of event-triggered real-time systems can be improve in order to bring it closer to that of a time-triggered real-time system.

6.6.2 Reducing Test Effort by Selecting Test Cases

Test effort can be reduced by omitting certain test cases. One such method is category partitioning [ABC82, BG75, OB87]. The basic idea is to split the input space into equivalence partitions and select representative cases from each partition. The assumption made is that any element of partition will expose errors as well as any other.

The fundamental difference between category partitioning and the method used in this work is that category partitioning reduces the effort at the cost of test coverage. The method discussed in this work aims at reducing test effort while still maintaining full test coverage.

Chapter 7

Conclusions

A summary of the work is presented in this chapter. The problems at hand, methods for solving them, and the results are discussed. Contributions that this work has made are also presented. Finally, future work is discussed.

7.1 Summary

This work is motivated by: i) making event-triggered real-time systems more testable is necessary to achieve the desired level of confidence in correctness of the system; and ii) investigating the feasibility of applying constraints that improve testability of event-triggered real-time systems is needed. Testability of real-time systems is mainly influenced by: test effort, observability, and reproducibility. This work concentrates on the test effort. The focus is on system level testing of timeliness, i.e., testing the entire system to see if deadlines are met. The main problems faced are: i) validation and refinement of existing constraints for improving testability; ii) investigation of effects that the constraints have; iii) provision of guidelines for constraint value selection; and iv) enforcement of the constraints. The overall method is to define a model for system level testing of real-time applications that are constrained in order to improve testability. This model is then used to perform validation of an existing upper bound on the test effort for real-time systems. Effects that the constraints have on predictability, efficiency, and scheduling are also investigated in the context of this model. Ways to enforce the constraints are discussed with respect to the model.

7.1.1 Validation and Refinement of Formulas

In work by Mellin, an upper bound on the test effort for event-triggered real-time systems is presented. The upper bound is formulated by constraining behavior of the system. Three constraints are defined: i) a maximum on the number of time a transaction can be preempted, ii) a maximum on the number of concurrently executing transactions triggered by the same event-type, and iii) points in time where the environment can be observed are constrained. A validation of this upper bound has been performed. The original upper bound has been corrected and refined. The refinement consisted of extending the upper bound so that it holds for a system with more than one resource.

The upper bound is high, therefore, ways to further reduce the test effort have been investigated. By only testing hard transactions completely the test effort can be reduced. Timeliness of hard transactions is of utmost importance while soft transactions may occasionally miss their deadlines. Therefore, it is possible to not fully test soft transactions. The approach taken is leave out execution states where soft transactions are not preempted maximum number of times. Instead of testing all combinations of soft and hard transactions only combinations where the soft transaction is not preempted or preempted maximum number of times are included. The reason is that these are the only combinations that can effect the execution of hard transactions. The situation where a soft transaction is not preempted can be considered as representative for situations where that transaction is preempted less than the allowed maximum. When a soft transaction has reached maximum preemptions a separate test case is needed because now the soft transaction can potentially delay execution of a hard transaction.

7.1.2 Effects on Real-Time Properties and Scheduling

The effects of constraining an event-triggered real-time system need investigation. These must be investigated with respect to selected real-time properties and system services. Part of the problem is therefore selection of appropriate properties and systems services. The selection of properties was constrained to predictability and efficiency. Selection of

predictability and efficiency is motivated by their role in achieving timeliness. Scheduling is a fundamental service in a real-time systems and plays an important role in achieving timeliness. It can be sensitive to any constraints placed on system behavior.

The effects of constraining preemptions, concurrent instances triggered by the same type, and observation points have been investigated on predictability, efficiency, and scheduling. The negative effects that these have on predictability and efficiency have been pointed out and means to counter them presented. Predictability is affected by constraining preemptions and concurrent instances. To maintain predictability there must exist a bound on the time to abort soft transactions and transaction instances cannot have different criticality. Efficiency is effected by the need to abort soft transaction is they reach maximum preemptions. The effort in aborting and restarting soft transactions decreases efficiency.

Further, effects on scheduling have been discussed. It has been concluded that preemption points must be used to constrain the points in time when a transaction can be preempted. Otherwise, a transaction can reach maximum allowed preemptions at an arbitrary point in its execution which makes it impossible to guarantee timeliness. Applying imprecise computation techniques has also been suggested as alternative to preemption points.

7.1.3 Guidelines for Constraint Value Selection

Selection of constraint values must be done in such a way that an application that uses the constraints to improve testability is still feasible. Deadlines should not be missed because constraints are introduced. To ensure that requires guidelines that condition constraint values selection. In this work necessary and sufficient conditions for constraint value selection for an application where all deadlines are hard, arrivals are periodic or sporadic, and deadlines are within respective periods have been produced. Using these guidelines it can be guaranteed that all deadlines that can be met with an unconstrained system will be met under the constraints.

7.1.4 Enforcement of Constraints

To realize the usage of constraints for improving testability methods for enforcement, i.e., making sure that constraints are not violated during execution, must exist. This can be done in three ways: i) before run-time the system is conditioned in such a way that violation cannot occur; ii) during run-time mechanisms are employed that ensure that constraints are not violated; and iii) constraint violations are detected during run-time and some recovery action executed. The applicability of each method has been investigated. Preventing constraint violations by conditioning the system is possible but demands restrictions of system behavior that are not always applicable. Avoiding constraint violation through run-time mechanisms is also possible. Two run-time mechanisms, adaptive filters and admission control, have been suggested. The main motivation for selecting these is that they have been investigated in the literature for different purposes and constraint violation handling can easily be integrated into these mechanisms.

Detection and recovery in the traditional sense is not applicable for enforcing the constraints. Traditionally an invalid state of the system is detected and through a recovery mechanism a safe state is found. In this work, if the constraints are violated the system cannot be returned to a safe state. The reason for this is that the upper bound defines the number of tests that have to be run in order to fully test the system. A violation of the constraints, e.g., preempting a task that has reached maximum preemptions, means that no such guarantee can be given because the execution of the system is no longer inside the scope defined by the upper bound. Therefore, the only recovery action taken is to notify the application that a violation has occurred. The scheduler has been selected as the most appropriate place to detect violations of maximum preemptions. The primary reason is that the scheduler is a component that is already in place in any real-time system and has access to information about transactions. For enforcing concurrent transaction instances the dispatcher cooperates with a special watchdog process that detects violations. This is motivated by the fact that the dispatcher has knowledge about when transactions are started and terminated. The actual detection is left to a separate process in order to reduce the workload on the dispatcher.

7.2 Contributions

In the following sections contributions that this work has made are presented.

7.2.1 Model for Using Constraints to Improve Testability

This work has defined a model for event-triggered real-time systems that are subject to constraints for improving testability. The purpose of this model is to define necessary properties that a system that uses the constraints to improve testability. The advantages of defining this model are: i) applicability of using constraints to improve testability is more easily determined because all the pertinent assumptions are clearly stated, and ii) necessary requirements on applications that use the constraints are stated. This model relies on some important properties: i) requirement on unique priorities; ii) requirement on acquiring all locks at the start of execution; and iii) requirement on exclusive locks.

7.2.2 Refinement of Upper Bound Test Effort

A refined upper bound on test effort for event-triggered real-time systems has been presented in this work. The main contribution is extension of formulation for potential blockings. The upper bound given be Mellin only handles a single resource being locked at a time. This has been extended to handle more than once lock at a time.

7.2.3 Effects of Applying Constraints to Improve Testability

An evaluation of effects that constraints to improve testability of event-triggered systems have on predictability, efficiency, and scheduling has been performed. Moreover, way to count these effects have been presented. Investigating effects on predictability, efficiency, scheduling is relevant because of their role in achieving timeliness.

7.3 Future Work

Several theoretical and practical problems that remain unsolved are presented in the following sections.

7.3.1 Refine Formula for Potential Blockings

To achieve predictability in a real-time database system there must be a bound on delay caused by blockings. The priority inheritance ceiling protocol is an example of a method that bounds this time. The formulas in this work assume arbitrary blockings. A more realistic approach would be to model the potential blockings under the priority ceiling protocol or a similar protocol.

7.3.2 Refinement of Guidelines

In this work guidelines for selection of constraint values have been suggested. Necessary sufficient conditions for constraint value selection where only given for a system with hard deadlines, and periodic or sporadic arrivals where deadlines are within respective periods. Sufficient conditions for the remaining situations given in section 5.2 are needed. Further, a more solid ground for selection of constraints values could be devised by performing an empirical experiment. The focus of such an experiment should be on feasibility of value selections to determine a criteria for different situations. Also, investigation of effects on performance would be useful.

7.3.3 Realization of Using Constraints that Improve Testability

To realize the use of the constraints programming constructs and implementation of support mechanisms is needed. The application programmer must be able to specify constraint values. Further, mechanisms that enforce the constraints need implementation. This includes implementing constraint enforcement into mechanisms such as admission

control and adaptive filers, extending scheduler to handle enforcement of preemption constraint, and implementing a mechanisms to enforce observation points.

7.3.4. Reducing the Test Effort Further

Even with the refinements suggested in this work the upper bound on test effort for event-triggered real-time system is prohibitively high. Additional investigation into ways to bring the bound down is needed.

7.3.5. Effects on Extensibility

Extensibility is an interesting property that needs investigation with respect to the constraints discussed in this dissertation. Extensibility is an essential property of any system. Comparing the extensibility of an unconstrained system to that of a constrained one is an interesting subject. A system that is constrained as discussed in this work is potentially more extensible than an unconstrained one. The reason being that the effort to validate changes is reduced while the effort associated with validating the constraints may often be negligible.

Acknowledgments

First of all, I would like to thank my supervisor Jonas Mellin for all his advise, support, criticism, and encouragement during this work. I would also like to thank Prof. Sten F. Andler for invaluable discussions and comments on this dissertation. Thanks to Esa Falkenroth and Ægir Leifsson for taking the time to review this dissertation and providing valuable comments. Ingi Sigurðsson should be thanked for all his help during my studies at the University of Skövde. Thanks to my family and friends. Finally, I would like to thank my late grandmother Hjördis Kristóferdóttir. Her unconditional love and spirit will always stay with me.

Bibliography

[AHEM95] Andler, S.F., et. al. *DeeDS - distributed active real-time database system*. Technical Report HS-IDA-TR-95-008. Department of computer Science University of Skövde. 1995.

[AHE+96] Andler, S.F., Hanson, J., Eriksson, J., Mellin, J., Berndtsson, M., and Eftring, B. *Deeds Towards a Distributed and Active Real-Time Database system*. ACM SIGMOD Record, Special issue on real-time database systems, 1996.

[Bei90] Beizer, B. *Software Testing Techniques*. (2nd ed.). Van Nostrand Reinhold, 1990

[Ber91] Berndtsson, M. *ACOOD: An approach to an active object-oriented DBMS*. Master's thesis, Department of Computer Science, University of Skövde. 1991.

[BH95] Berndtsson, M., and Hansson, J. *Issues in active real-time databases*. Proceedings of Active and Real-Time Database Systems (ARTDB-95) Editors: Berndtsson, M., and Hansson, J. Springer Verlag. 1995.

[BN97] Bestavros, A., and Nagy, S. *Admission control and overload management for real-time databases*. Real-Time database systems - Issues and applications. Eds. A. Bestavros, K-J Lin, S.H. Son. Kluwer academic publishers. 1997.

[Bur91] Burns, A. *Scheduling Hard Real-Time Systems*. Software Engineering Journal. May 1991.

[CBB+89] Chakravarthy, S., Blaustein, A., Buckman, A., Carey, M.J., Dayal, U., Goldhirsch, M., Hsu, R., Juahari, R., Ladin, M., Livny, D., MacCarthy, R., MacKee, R., Rosenthal, A. *HiPAC: A research project in active time-constrained database managment*. Final technical report. Technical report XAIT-89-02, Xerox advanced information technology, August 1989.

[CS87] Cheng, S.-C., and Stankovic, J.A. *Scheduling algorithms for hard real-time systems - a brief survey*. In Stankovic, J.A., editor, Hard Real-Time Systems. IEEE Computer Society Press. 1987.

[Dei90] Deitel, H.M. *Operating Systems*(2nd ed.). Addison-Wesley. 1990.

[Duk89] Duke, E.L. *V&V of flight and mission-critical software*. IEEE Software, vol.6, pp. 39-35, May 1989.

[EN94] Elmasri, R., and Navathe, S. *Fundamentals of database systems (2nd edition)*. Benjamin/Cummings, 1994.

[Gai85] Gait, J. *A debugger for concurrent programs*, Software - Practice and experience, Vol 15. June 1985

[GGD91] Gatziu, S., Geppert, A., and Dittrich, K.R. *Integrating active concepts into an object-oriented database system*. Proceedings of the third international workshop on database programming language. Nafplion. Greece. August 1991.

[GHS+95] Goya, B., Haritsa, J., Seshadri, S., and Srinivasan, V. *Index concurrency control in firm real-time dbms*. Proceedings of the 21st VLDB conference, pp. 146-157, September 1995.

[Gla80] Glass, R. L. *Real-Time: The "lost world" of software debugging and testing*. Communications of the ACM, vol. 23, May 1980.

[GR93] Gray, J., and Reuter, A., *Transaction processing: Concepts and techniques*. Morgan Kaufman, 1993.

[Han94] Hansson, J. *Dynamic real-time transaction scheduling with multiple combined performance metrics*. MSc Thesis, University Of Skövde, 1993.

[Han89] Hanson, E. N. *An initial report on the design of Ariel: A DBMS with an integrated production rule system*. SIGMOD Record, special issue on rule management and processing in expert database systems. September 1983.

[HCL90] Haritsa, J.R., Linvy, M., and Carey, M.J. *Earliest deadline first scheduling for real-time database systems*. Proceedings of the 12th real-time systems symposium, December 1991.

[Het88] Hetzel, B. *The complete guide to software testing*(2nd ed.). QED information science. Wellesley. 1988.

[Hor74] Horn, W.A. *Some simple scheduling algorithms*. Naval Research Logistics Quarterly, 21, 1974.

[Kop91] Kopetz, H. *Event-triggered versus Time-Triggered real-time systems*, Int. workshop on operating systems of the 90s and beyond, Springer-Verlag, 1991.

[Kop93] Kopetz, H., Foreword, The testability of distributed real-time systems by Schütz, W. Kluwer Academic Publishers. 1993.

[Kop93] Koperz, H. *Real-Time Scheduling (Chapter 18)*. Distributed Systems(2nd edition). Editor S. Mullender. Addison-Wesley. 1993.

[Lap94] Laprie et. al. *Dependability: Basic concepts and terminology*. WG 10.4. IFIP. 1994.

[LC86] Liestman, A. L., and Campbell, R. H.. *A fault tolerant scheduling problem*. IEEE transactions on software engineering, vol. SE-12, No. 10, pp. 1089- 1095, October 1986.

[LLY+91] Lie, J.W.S., Lin, K-J., Shi, W.-K., Yu, A. C.-S., Chung, J.-Y., and Zhao, W. IEEE Computer, vol, 24, no. 5, May 1991, pp. 58-68.

[LL73] Liu, C.L., and Layland, J., *Scheduling algorithms for multiprogramming in a hard real-time environment*. Journal of the Association for Computer Machinery, February 1973, pp. 46-61.

[LNL87] Lin, K., Natarajan, S., and Liu, J. *Imprecise results: Utilizing partial computations in real-time systems.* Proceedings of IEEE real-time systems symposium. 1987.

[Mel96] Mellin, J. *Supporting system level testing of applications by active real-time database systems*. To appear in: Proceedings of the second International Workshop on Active, Real-Time, and Temporal Database Systems. Springer-Verlag. 1997.

[MHA97] Mellin, J, Hanson, J., and Andler, S.F. *Refining timing constraints of applications in DEEDS*. Real-Time Database Systems: Issues and Applications. Kluwer Academic Publishers, 1997.

[Mar82] Marte, C. *Preemptive scheduling with release times, deadlines, and due times*. Journal of the Association for Computer Machinery, Mars 1982.

[Mye79] Myers, G.J. *The art of software testing*. John Wiley and Sons. New York. 1979.

[Par90] Parnas, D. L., Schouwen, A. J., and Kwan, S. P. *Evaluation of safety-critical software*. Communications of the ACM, vol. 33, pp. 636-648, June 1990.

[RLK+95] Randell, B., Laprie, J.-C., Kopetz, H., and Littelwood, B., eds. *Predictably Dependable Computing Systems*. Springer-Verlag. 1995.

[Rat93] Ramamritham, K. Real-Time Databases. *Distributed and Parallel Databases*, vol. 1, 1993, p. 199-226.

[RSS90] Ratmaritham, K., Stankovic, J., and Shiah, P. *Efficient scheduling algorithms for real-time multiprocessor systems*. IEEE Transactions on Parallel and distributed computing, Vol. 2, pp. 184-194, April 1990.

[Sch93] Schütz, W. *The testability of distributed real-time systems*. Kluwer Academic Publishers. 1993.

[SJG+90] Stonebreaker, M., Jhingran, M., Goth, J., and Potamianos, S., *On rules, procedures, caching, and views in data base systems*. In proceedings of the ACM SIGMOD international conference on management of data, pp. 281-290, New Jersey, May 1990.

[SL86] Lehoczky, J. P., and Sha, L. *Performance of bus scheduling algorithm*. In Performance 86, 1986.

[Som92] Sommerville, I. *Software Engineering* (4th ed.). Addison-Wesley. 1992.

[SRL90] Sha, L., Rajkumar, R., and Lehoczky, J.P. Priority inheritance protocols: An approach to real-time synchronization. IEEE transactions on computers, Vol. 39, No.9 , September 1990, pp. 1175-1185.

[Sta90] Stankovic, J.A., and Ramamritham, K. "*What is predictability for Real-Time Systems?*". Real-Time Systems, Vol. 2, Dec. 1990, p. 247-254. Kluwer Academic Publishers.

[Sta88] Stankovic, J.A.. *Misconcepts about Real-Time Computing : A serious problem for next-generation systems*. IEEE Computer, Vol. 21, Okt. 1988, p. 10-19.

[SR95] Stankovic, J.A., and Ramamritham, K. *A reflective architecture for real-time operating systems*. Advances in real-time systems. Editor S.H. Son. Prentice-Hall. 1995.

[SR93] Stankovic, J., and Ramamritham, K.. *The Spring kernel: A new paradigm for real-time systems. IEEE Software, Vol. 8, No. 3, May 1991, pp. 62-72.*

[Sta95] Stankovic, J. *Adjustable flow control filters, reflective memories, and coprocessors as support for distributed real-time systems*. International journal of mini an microcomputers. Vol. 17, No.1, 1995.

[VK93] Verissimo, P., and Kopetz, H. *Real-Time and dependability concepts*. Distributed Systems(2nd edition). Editor S. Mullender. Addison-Wesley. 1993.

[Vrb91] Vrbsky, S., and Liu, J.W.S. *An object oriented query processor that returns monotonically improving answers*. Proceedings of seventh IEEE international conference on data engineering. IEEE CS Press. Los Alamitos, California. 1991.

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

Appendix A - ESTAT and ESTAT´

A verification and comparison of ESTAT and ESTAT´ is presented in this appendix. Four examples are given and the number of states calculated using ESTAT (section A.1) and ESTAT´ (section A.2) respectively. The objective with this is: i) to show that the formulas calculate a correct number of states, and ii) illustrate the difference between ESTAT and ESTAT´. For each table $S_1..S_n$ is an observation interval. An event is represented by a letter, e.g., a, b, or c. The non occurrence of an event is represented by 0. For example, in table A.1.1 two events, a and b, can occur in the single observation interval S_1.

A.1 - ESTAT

No.	S_1
1	0
2	a
3	b
4	ab

Table A.1.1 - s=1, n=2

$ESTAT(s,n)=(s+1)^n=(1+1)^2=2^2=4$

No.	S_1	S_2
1	0	0
2	a	0
3	0	a
4	b	0
5	0	b
6	a	b
7	b	a
8	ab	0
9	0	ab

Table A.1.2 - s=2, n=2

$ESTAT(s,n)=(s+1)^{n=}\ (2+1)^2=3^2=9$

No.	S1	S2	S3
1	0	0	0
2	a	0	0
3	0	a	0
4	0	0	a
5	b	0	0
6	0	b	0
7	0	0	b
8	a	b	0
9	a	0	b
10	0	a	b
11	b	a	0
12	b	0	a
13	0	b	a
14	ab	0	0
15	0	ab	0
16	0	0	ab

Table A.1.3 - s=3, n=2

ESTAT(s,n)=$(s+1)^{n=}(3+1)^2=4^2=16$

No.	S_1	S_2	S_3	No.	S_1	S_2	S_3
1	0	0	0	27	c	0	b
2	a	0	0	28	0	c	b
3	0	a	0	29	ab	0	0
4	0	0	a	30	0	ab	0
5	b	0	0	31	0	0	ab
6	0	b	0	32	ac	0	0
7	0	0	b	33	0	ac	0
8	c	0	0	34	0	0	ac
9	0	c	0	35	cb	0	0
10	0	0	c	36	0	cb	0
11	a	b	0	37	0	0	cb
12	a	0	b	38	a	b	c
13	0	a	b	39	a	c	b
14	a	c	0	40	b	a	c
15	a	0	c	41	b	c	a
16	0	a	c	42	c	a	b
17	b	c	0	43	c	b	a
18	b	0	c	44	ab	c	0
19	0	b	c	45	ab	0	c
20	b	a	0	46	0	ab	c
21	b	0	a	47	ab	c	0
22	0	b	a	48	ab	0	c
23	c	a	0	49	0	ab	c
24	c	0	a	50	ac	b	0
25	0	c	a	51	ac	0	b
26	c	b	0	52	0	ac	b

53	bc	a	0	59	b	ac	0
54	bc	0	a	60	b	0	ac
55	0	bc	a	61	0	b	ac
56	a	bc	0	62	c	ab	0
57	a	0	bc	63	c	0	ab
58	0	a	bc	64	0	c	ab

Table A.1.4 - s=3, n=3

$ESTAT(s,n)=(s+1)^{n=}(3+1)^3=4^3=64$

A.2 - ESTAT´

As discussed in section 4.4 ESTAT´ reduces the number of observable environmental states by omitting cases where only non-critical events arrive. In ESTAT´ h represents the number of critical events out of the n events that can arrive. Criticality of event types is given in each table. For example in table A.2.1 events of two types, a and b, can arrive. A is a critical event type while b is not.

No.	S_1
1	0
2	a
4	ab

Table A.2.1 - s=1, n=2, h=1 critical events {a} ; non critical events {b}

$$ESTAT'(s,n,h)=(s+1)^n-((s+1)^{n-h}-1)=(1+1)^2-((1+1)^1-1)=4-1=3$$

No.	S_1	S_2
1	0	0
2	a	0
3	0	a
4	a	b
5	b	a
6	ab	0
7	0	ab

Table A.2.2 - s=2, n=2, h=1 critical events {a} ; non critical events {b}

$$ESTAT'(s,n,h) = (s+1)^n - ((s+1)^{n-h} - 1) = (2+1)^2 - ((2+1)^1 - 1) = 9 - 2 = 7$$

No.	S_1	S_2	S_3
1	0	0	0
2	a	0	0
3	0	a	0
4	0	0	a
5	a	b	0
6	a	0	b
7	0	a	b
8	b	a	0
9	b	0	a
10	0	b	a
11	ab	0	0
12	0	ab	0
13	0	0	ab

Table A.2.3 s=3, n=2, h=1 critical events {a} ; non critical events {b}

$$ESTAT'(s,n,h) = (s+1)^n - ((s+1)^{n-h} - 1) = (3+1)^2 - ((3+1)^1 - 1) = 16 - 3 = 13$$

No.	S_1	S_2	S_3
1	0	0	0
2	a	0	0
3	0	a	0
4	0	0	a
5	a	b	0
6	a	0	b
7	0	a	b
8	a	c	0
9	a	0	c
10	0	a	c
11	b	a	0
12	b	0	a
13	0	b	a
14	c	a	0
15	c	0	a
16	0	c	a
17	ab	0	0
18	0	ab	0
19	0	0	ab
20	ac	0	0
21	0	ac	0
22	0	0	ac
23	a	b	c
24	a	c	b
25	b	a	c
26	b	c	a
27	c	a	b
28	c	b	a

29	ab	c	0
30	ab	0	c
31	0	ab	c
32	ab	c	0
33	ab	0	c
34	0	ab	c
35	ac	b	0
36	ac	0	b
37	0	ac	b
38	bc	a	0
39	bc	0	a
40	0	bc	a
41	a	bc	0
42	a	0	bc
43	0	a	bc
44	b	ac	0
45	b	0	ac
46	0	b	ac
47	c	ab	0
48	c	0	ab
49	0	c	ab

Table A.2.4 - s=3, n=3, h=2 critical events {a} ; non critical events {b,c}

$$ESTAT'(s,n,h) = (s+1)^n - ((s+1)^{n-h} - 1) = (3+1)^3 - ((3+1)^2 - 1) = 64 - 15 = 49$$

Appendix B - PSTAT´ and PSTAT´´

A verification and comparison of PSTAT´ and PSTAT´´ is presented in this appendix. In section B.1 seven examples are given and the number of states calculated using PSTAT´. In section B.2 two examples are given for PSTAT´´. The examples given for PSTST´´ are the same ones as in tables B.1.6 and B.1.7. As discussed in section 4.4 PSTAT´´ requires that p>=1. Therefore, tables B.1.1, B.1.3, and B.1.6 do not apply for PSTAT´´. The objective of this is: i) to show that the formulas calculate a correct number of states, and ii) illustrate the difference between PSTAT´ and PSTAT´´. For each table $T_1..T_n$ are transaction instances triggered by the same type. Transaction instances triggered by different types are represented by $T_a..T_z$. For example, in Table B.1 a single transaction instance T_1 can execute and is allowed to be preempted at most once. Three possibilities exist: T_1 does not execute (#), T_1 executes and is not preempted (0), and T_1 executes and is preempted once (1).

B.1 - PSTAT´

No.	T_1
1	#
2	0
3	1

Table B.1.1 - p=1, q=1, t=1

$$PSTAT\ ´(p,q,t) = (\sum_{k=0}^{q}(p+1)^k)^t = \ (\frac{(1+1)^{1+1}-1}{1})^1 = \frac{2^2-1}{1} = 3$$

This examples illustrates that PSTAT´ calculates a upper bound on the number of execution states. In this example we have one event type which triggers a single transaction instance. Therefore, zero or one transactions can execute, and it is either

preempted or not. However, a transaction has to be preempted in favor of some other transaction. Therefore, the execution state where preemption occurs is not possible.

Nr.	T_1
1	#
2	0
3	1
4	2

Table B.1.2 - p=2, q=1, t=1

$$PSTAT\ ´(p,q,t) = (\sum_{k=0}^{q}(p+1)^k)^t = (\frac{(2+1)^{1+1}-1}{2})^1 = \frac{3^2-1}{2} = \frac{8}{2} = 4$$

No.	T_1	T_2
1	#	#
2	0	#
3	1	#
4	0	0
5	1	0
6	0	1
7	1	1

Table B.1.3 - p=1, q=2, t=1

$$PSTAT\ ´(p,q,t) = (\sum_{k=0}^{q}(p+1)^k)^t = (\frac{(1+1)^{2+1}-1}{1})^1 = \frac{2^3-1}{1} = \frac{7}{1} = 7$$

No.	T_1	T_2
1	#	#
2	0	#
3	1	#
4	2	#
5	0	0
6	1	0
7	2	0
8	0	1
9	1	1
10	2	1
11	0	2
12	1	2
13	2	2

Table B.1.4 - p=2, q=2, t=1

$$PSTAT\ ´(p,q,t) = (\sum_{k=0}^{q}(p+1)^k)^t = (\frac{(2+1)^{2+1}-1}{2})^1 = \frac{3^3-1}{2} = \frac{26}{2} = 13$$

No.	Ta	Tb
1	#	#
2	#	0
3	#	1
4	#	2
5	0	#
6	0	0
7	0	1
8	0	2
9	1	#
10	1	0
11	1	1
12	1	2
13	2	#
14	2	0
15	2	1
16	2	2

Table B.1.5 - p=2, q=1, t=2

$$PSTAT\ ´(p,q,t) = (\sum_{k=0}^{q}(p+1)^k)^t = (\frac{(2+1)^{1+1}-1}{2})^2 = (\frac{3^2-1}{2})^2 = 4^2 = 16$$

No.	Ta	Tb	Tc
1	#	#	#
2	#	#	0
3	#	#	1
4	#	0	#
5	#	0	0
6	#	0	1
7	#	1	#
8	#	1	0
9	#	1	1
10	0	#	#
11	0	#	0
12	0	#	1
13	0	0	#
14	0	0	0
15	0	0	1
16	0	1	#
17	0	1	0
18	0	1	1

19	1	#	#
20	1	#	0
21	1	#	1
22	1	0	#
23	1	0	0
24	1	0	1
25	1	1	#
26	1	1	0
27	1	1	1

Table B.1.6 - p=1, q=1, t=3

$$PSTAT\ ´(p,q,t) = (\sum_{k=0}^{q}(p+1)^k)^t = (\frac{(1+1)^{1+1}-1}{1})^3 = (\frac{2^2-1}{1})^3 = 3^3 = 27$$

No.	Ta	Tb	Tc	No.	Ta	Tb	Tc
1	#	#	#	33	1	#	#
2	#	#	0	34	1	#	0
3	#	#	1	35	1	#	1
4	#	#	2	36	1	#	2
5	#	0	#	37	1	0	#
6	#	0	0	38	1	0	0
7	#	0	1	39	1	0	1
8	#	0	2	40	1	0	2
9	#	1	#	41	1	1	#
10	#	1	0	42	1	1	0
11	#	1	1	43	1	1	1
12	#	1	2	44	1	1	2
13	#	2	#	45	1	2	#
14	#	2	0	46	1	2	0
15	#	2	1	47	1	2	1
16	#	2	2	48	1	2	2
17	0	#	#	49	2	#	#
18	0	#	0	50	2	#	0
19	0	#	1	51	2	#	1
20	0	#	2	52	2	#	2
21	0	0	#	53	2	0	#
22	0	0	0	54	2	0	0
23	0	0	1	55	2	0	1
24	0	0	2	56	2	0	2
25	0	1	#	57	2	1	#
26	0	1	0	58	2	1	0
27	0	1	1	59	2	1	1
28	0	1	2	60	2	1	2
29	0	2	#	61	2	2	#
30	0	2	0	62	2	2	0
31	0	2	1	63	2	2	1
32	0	2	2	64	2	2	2

Table B.1.7 - p=2, q=1, t=3

$$PSTAT\ ´(p,q,t) = (\sum_{k=0}^{q}(p+1)^k)^t = (\frac{(2+1)^{1+1}-1}{2})^3 = (\frac{3^2-1}{2})^3 = \left(\frac{8}{2}\right)^3 = 4^3 = 64$$

B.2 - PSTAT´´

As discussed in section 4.4 PSTAT´´ reduces the number execution states by omitting cases where soft transaction instances are preempted less than maximum number of times. In PSTAT´´ the variable w represents the number of critical event types out of t events types that exist. Criticality of event types that trigger the transactions is given in each table. For example in table B.2.1 transactions triggered by events of two types, a and b, can arrive. T_a is triggered by a critical event type while T_b is not.

No.	T_1
1	#
2	0
3	2

Table B.2.1 - p=2, q=1, t=1 transaction T_1 is triggered by a critical event type.

$$PSTAT\ ´´(p,q,t,w) = \left(\sum_{k=0}^{q}(p+1)^k\right)^w * \left(\left(\sum_{k=0}^{q}(p+1)^k\right) - \left(\sum_{k=0}^{q}p^k\right)\right)^{t-w} =$$

$$= (\frac{((2+1)^{1+1}-1}{2})^0 * ((\frac{(2+1)^{1+1}-1}{2}) - (\frac{2^{1+1}-1}{1}) + 2)^{1-0} =$$

$$= 1 * ((\frac{3^2-1}{2})^2 - (\frac{2^2-1}{1}) + 2)^1 = 1 * (4-3+2)^1 = 3$$

No.	T_a	T_b
1	#	#
2	#	0
3	#	2
4	0	#
5	0	0
6	0	2
7	1	#
8	1	0
9	1	2
10	2	#

11	2	0
12	2	2

Table B.2.2 - p=2, q=1, t=2, w=1 critical events {a} ; non critical events {b}

$$PSTAT\,''(p,q,t,w) = \Big(\sum_{k=0}^{q}(p+1)^k\Big)^{w} * \Big(\Big(\sum_{k=0}^{q}(p+1)^k\Big) - \Big(\sum_{k=0}^{q}p^k\Big)\Big)^{t-w} =$$

$$= \Big(\frac{((2+1)^{1+1}-1}{2}\Big)^1 * \Big(\Big(\frac{(2+1)^{1+1}-1}{2}\Big) - \Big(\frac{2^{1+1}-1}{1}\Big) + 1 + 1\Big)^{2-1} =$$

$$= 4 * \Big(\Big(\frac{3^2-1}{2}\Big)^2 - \Big(\frac{2^2-1}{1}\Big) + 2\Big)^1 = 4 * (4-3+2)^1 = 4 * 3 = 12$$

No.	T_1	T_2
1	#	#
2	0	#
3	2	#
4	0	0
5	2	0
6	0	2
7	2	2

Table B.2.3 - p=2, q=2, t=1 both transactions are triggered by a critical event type

$$PSTAT\,''(p,q,t,w) = \Big(\sum_{k=0}^{q}(p+1)^k\Big)^{w} * \Big(\Big(\sum_{k=0}^{q}(p+1)^k\Big) - \Big(\sum_{k=0}^{q}p^k\Big)\Big)^{t-w} =$$

$$= \Big(\frac{((2+1)^{2+1}-1}{2}\Big)^0 * \Big(\Big(\frac{(2+1)^{2+1}-1}{2}\Big) - \Big(\frac{2^{2+1}-1}{1}\Big) + 2\Big)^{1-0} =$$

$$= 1 * \Big(\Big(\frac{3^3-1}{2}\Big)^2 - \Big(\frac{2^3-1}{1}\Big) + 3\Big)^1 = 1 * (13-8+2)^1 = 7$$

No.	Ta	Tb	Tc
1	#	#	#
2	#	#	0
3	#	#	2
4	#	0	#
5	#	0	0
6	#	0	2
7	#	2	#
8	#	2	0
9	#	2	2

10	0	#	#
11	0	#	0
12	0	#	2
13	0	0	#
14	0	0	0
15	0	0	2
16	0	2	#
17	0	2	0
18	0	2	2
19	1	#	#
20	1	#	0
21	1	#	2
22	1	0	#
23	1	0	0
24	1	0	2
25	1	2	#
26	1	2	0
27	1	2	2
28	2	#	#
29	2	#	0
30	2	#	2
31	2	0	#
32	2	0	0
33	2	0	2
34	2	2	#
35	2	2	0
36	2	2	2

Table B.2.4 - p=2, q=1, t=3, w=1 critical events {a} ; non critical events {b,c}

$$PSTAT\,''(p,q,t,w) = \left(\sum_{k=0}^{q}(p+1)^k\right)^{w} * \left(\left(\sum_{k=0}^{q}(p+1)^k\right) - \left(\sum_{k=0}^{q}p^k\right)\right)^{t-w} =$$

$$\left(\frac{(2+1)^{1+1}-1}{2}\right)^{1} * \left(\left(\frac{(2+1)^{1+1}-1}{2}\right) - \left(\frac{2^{1+1}-1}{1}\right)+1+1\right)^{3-1}$$

$$= \left(\frac{3^2-1}{2}\right)^{1} * \left(\left(\frac{3^2-1}{2}\right)^{2} - \left(\frac{2^2-1}{1}\right)+2\right)^{2} = 4*(4-3+2)^2 = 4*3^2 = 36$$

Appendix C - BSTAT´ and BSTAT´´

A verification and comparison of BSTAT´ and BSTAT´´ is presented in this appendix. Four examples are given and the number of states calculated using BSTAT´ (section C.1) and BSTAT´´ (section C.2) respectively. The objective with this is: i) to show that BSTAT´´ calculates correct number of states, and ii) illustrate the difference between BSTAT ´ and BSTAT´´. The notation used is the same one presented in section 4.3.1.

C.1 - BSTAT´

No.	Blocking
1	a [b

Table C.1.1 - q=1, t=2, C={0,2}

$$BSTAT'(q,t,C) = \prod_{k=0}^{r-2} \binom{q*t - \sum_{l=0}^{k} c_l}{c_{k+1}} = \prod_{k=0}^{2-2} \binom{1*2 - \sum_{l=0}^{k} c_l}{c_{k+1}} = \binom{2}{2} = 1$$

No.	Blocking
1	a [b
2	a [c
3	b [c

Table C.1.2 - q=1, q=3, C={0,2}

$$BSTAT'(q,t,C) = \prod_{k=0}^{r-2} \binom{q*t - \sum_{l=0}^{k} c_l}{c_{k+1}} = \prod_{k=0}^{2-2} \binom{1*3 - \sum_{l=0}^{k} c_l}{c_{k+1}} = \binom{3}{2} = 3$$

No.	Blocking
1	a [b & c [d
2	a [c & b [d
3	a [d & b [c
4	b [c & a [d
5	b [d & a [c
6	c [d & a [b

Table C.1.3 - q=1, t=4, C={0,2,2}

$$BSTAT'(q,t,C) = \prod_{k=0}^{r-2} \binom{q*t - \sum_{l=0}^{k} c_l}{c_{k+1}} = \prod_{k=0}^{3-2} \binom{1*4 - \sum_{l=0}^{k} c_l}{c_{k+1}} = \binom{4}{2} * \binom{2}{2} = 6*1 = 6$$

No.	Blocking
1	a [bc & d [e
2	a [bc & d [f
3	a [bc & e [f
4	a [bd & c [e
5	a [bd & c [f
6	a [bd & c [e
7	a [be & c [d
8	a [be & c [f
9	a [be & d [f
10	a [bf & c [d
11	a [bf & c [e
12	a [bf & e [d
13	a [cd & b [e
14	a [cd & b [f
15	a [cd & e [f
16	a [ce & b [d
17	a [ce & b [f
18	a [ce & d [f
19	a [cf & b [d
20	a [cf & b [e
21	a [cf & e [d
22	a [de & b [c
23	a [de & b [f
24	a [de & c [f
25	a [df & b [c
26	a [df & b [e
27	a [df & c [f
28	a [ef & b [c
29	a [ef & b [d
30	a [ef & c [d
31	b [cd & a [e

32	b [cd & a [f
33	b [cd & e [f
34	b [ce & a [d
35	b [ce & a [f
36	b [ce & d [f
37	b [cf & a [d
38	b [cf & a [e
39	b [cf & d [e
40	b [de & a [c
41	b [de & a [f
42	b [de & c [f
43	b [df & a [c
44	b [df & a [e
45	b [df & c [e
46	b [ef & a [c
47	b [ef & a [d
48	b [ef & c [d
49	c [de & a [b
50	c [de & a [f
51	c [de & b [f
52	c [df & a [b
53	c [df & a [e
54	c [df & b [e
55	c [ef & a [b
56	c [ef & a [d
57	c [ef & b [d
58	d [ef & a [b
59	d [ef & a [c
60	d [ef & b [c

Table C.1.4 - q=1, t=6, C={0,3,2}

$$BSTAT´(q,t,C) = \prod_{k=0}^{r-2} \binom{q*t - \sum_{l=0}^{k} c_l}{c_{k+1}} = \prod_{k=0}^{3-2} \binom{1*6 - \sum_{l=0}^{k} c_l}{c_{k+1}} = \binom{6}{3} * \binom{3}{2} = 20*3 = 60$$

No.	Blocking
1	a [bc & d [ef
2	a [bd & c [ef
3	a [be & c [df
4	a [bf & c [de
5	a [cd & b [ef
6	a [ce & b [df
7	a [cf & b [de
8	a [de & b [cf
9	a [df & b [ce

10	a [ef & b [cd
11	b [cd & a [ef
12	b [ce & a [ef
13	b [cf & a [de
14	b [de & a [cf
15	b [df & a [ce
16	b [ef & a [cd
17	c [de & [bd
18	c [df & a [be
19	c [ef & a [bd
20	d [ef & a bc

Table C.1.5 - q=1, t=6, C={0,3,3}

$$BSTAT'(q,t,C) = \prod_{k=0}^{r-2} \binom{q*t - \sum_{l=0}^{k} c_l}{c_{k+1}} = \prod_{k=0}^{3-2} \binom{1*6 - \sum_{l=0}^{k} c_l}{c_{k+1}} = \binom{6}{3} * \binom{3}{3} = 20 * 1 = 20$$

C.2 - BSTAT´´

No.	Blocking
1	a [b

Table C.2.1 - q=1, t=2, C={0,2}, O={1}

$$BSTAT''(q,t,C,O) = \frac{\prod_{k=0}^{r-2} \binom{q*t - \sum_{l=0}^{k} c_l}{c_{k+1}}}{\prod_{m=1}^{n} o_m!} = \frac{\prod_{k=0}^{2-2} \binom{1*2 - \sum_{l=0}^{k} c_l}{c_{k+1}}}{\prod_{m=1}^{1} o_m!} = \frac{\binom{2}{2}}{1!} = \frac{2}{2} = 1$$

No.	Blocking
1	a [b
2	a [c
3	b [c

Table C.2.2 - q=1, q=3, C={0,2}, O={1}

$$BSTAT''(q,t,C,O) = \frac{\prod_{k=0}^{r-2} \binom{q*t-\sum_{l=0}^{k} c_l}{c_{k+1}}}{\prod_{m=1}^{n} o_m!} = \frac{\prod_{k=0}^{2-2} \binom{1*3-\sum_{l=0}^{k} c_l}{c_{k+1}}}{\prod_{m=1}^{1} o_m!} = \frac{\binom{3}{2}}{1!} = \frac{3}{2} = 3$$

No.	Blocking
1	a [b & c [d
2	a [c & b [d
3	a [d & b [c

Table C.2.3 - q=1, t=4, C={0,2,2}, O={2}

$$BSTAT''(q,t,C,O) = \frac{\prod_{k=0}^{r-2} \binom{q*t-\sum_{l=0}^{k} c_l}{c_{k+1}}}{\prod_{m=1}^{n} o_m!} = \frac{\prod_{k=0}^{3-2} \binom{1*4-\sum_{l=0}^{k} c_l}{c_{k+1}}}{\prod_{m=1}^{1} o_m!} = \frac{\binom{4}{2}*\binom{2}{2}}{2!} = \frac{6}{2} = 3$$

No.	Blocking
1	a [bc & d [e
2	a [bc & d [f
3	a [bc & e [f
4	a [bd & c [e
5	a [bd & c [f
6	a [bd & c [e
7	a [be & c [d
8	a [be & c [f
9	a [be & d [f
10	a [bf & c [d
11	a [bf & c [e
12	a [bf & e [d
13	a [cd & b [e
14	a [cd & b [f

15	a [cd & e [f
16	a [ce & b [d
17	a [ce & b [f
18	a [ce & d [f
19	a [cf & b [d
20	a [cf & b [e
21	a [cf & e [d
22	a [de & b [c
23	a [de & b [f
24	a [de & c [f
25	a [df & b [c
26	a [df & b [e
27	a [df & c [f
28	a [ef & b [c
29	a [ef & b [d
30	a [ef & c [d
31	b [cd & a [e
32	b [cd & a [f
33	b [cd & e [f
34	b [ce & a [d
35	b [ce & a [f
36	b [ce & d [f
37	b [cf & a [d
38	b [cf & a [e
39	b [cf & d [e
40	b [de & a [c
41	b [de & a [f
42	b [de & c [f
43	b [df & a [c
44	b [df & a [e
45	b [df & c [e
46	b [ef & a [c
47	b [ef & a [d
48	b [ef & c [d
49	c [de & a [b
50	c [de & a [f
51	c [de & b [f
52	c [df & a [b
53	c [df & a [e
54	c [df & b [e
55	c [ef & a [b
56	c [ef & a [d
57	c [ef & b [d
58	d [ef & a [b
59	d [ef & a [c
60	d [ef & b [c

Table C.1.4 - q=1, t=6, C={0,3,2}, O={1,1}

$$BSTAT''(q,t,C,O) = \frac{\prod_{k=0}^{r-2} \binom{q*t - \sum_{l=0}^{k} c_l}{c_{k+1}}}{\prod_{m=1}^{n} o_m!} = \frac{\prod_{k=0}^{3-2} \binom{1*6 - \sum_{l=0}^{k} c_l}{c_{k+1}}}{\prod_{m=1}^{1} o_m!} = \frac{\binom{6}{3} * \binom{3}{2}}{1! * 1!} = \frac{60}{1} = 60$$

No.	Blocking
1	a [bc & d [ef
2	a [bd & c [ef
3	a [be & c [df
4	a [bf & c [de
5	a [cd & b [ef
6	a [ce & b [df
7	a [cf & b [de
8	a [de & b [cf
9	a [df & b [ce
10	a [ef & b [cd

Table C.2.4 - q=1, t=6, C={0,3,3}, O={2}

$$BSTAT''(q,t,C,O) = \frac{\prod_{k=0}^{r-2} \binom{q*t - \sum_{l=0}^{k} c_l}{c_{k+1}}}{\prod_{m=1}^{n} o_m!} = \frac{\prod_{k=0}^{3-2} \binom{1*6 - \sum_{l=0}^{k} c_l}{c_{k+1}}}{\prod_{m=1}^{1} o_m!} = \frac{\binom{6}{3} * \binom{3}{3}}{2!} = \frac{20}{2} = 10$$

Appendix D - Test Case Example

Consider the following example where two event types E_1 and E_2 exists. Two transaction instances T_1 and T_2 triggered by E_1 and E_2 respectively exist. Each instance be at most be preempted once during its execution. Both instances need access to one resource, and T_1 has a higher priority than T_2. The following execution sequences are possible:

1) No transaction executes.
2) Only transaction T_1 executes and is not preempted.
3) Only transaction T_2 executes and is not preempted.
4) Both T_1 and T_2 execute and neither is preempted.
5) Both T_1 and T_2 execute and T_2 is preempted in favor T_1.

The following blockings are possible:

1) No blocking occurs.
2) Transaction T_2 blocks transaction T_1.

Assuming that the system is only observed once the following event arrivals are possible:

1) No event arrives
2) Only E_1 arrives
3) Only E_2 arrives
4) E_1 and E_2 arrive

Using the above information the following test cases are needed:

No.	Execution	Preemption	Blocking	Request arrival
1	None	None	None	None
2	T_1	None	None	None
3	T_1	None	None	E_1
4	T_1	None	None	E_2
5	T_1	None	None	E_1 & E_2
6	T_2	None	None	None
7	T_2	None	None	E_1
8	T_2	None	None	E_2
9	T_2	None	None	E_1 & E_2
10	T_1 & T_2	None	None	None
11	T_1 & T_2	None	None	E_1
12	T_1 & T_2	None	None	E_2
13	T_1 & T_2	None	None	E_1 & E_2
14	T_1 & T_2	None	T2 [T1	None
15	T_1 & T_2	None	T2 [T1	E_1
16	T_1 & T_2	None	T2 [T1	E_2
17	T_1 & T_2	None	T2 [T1	E_1 & E_2
18	T_1 & T_2	T2	No	None
19	T_1 & T_2	T2	No	E_1
20	T_1 & T_2	T2	No	E_2
21	T_1 & T_2	T2	No	E_1 & E_2
22	T_1 & T_2	T2	T2 [T1	None
23	T_1 & T_2	T2	T2 [T1	E_1
24	T_1 & T_2	T2	T2 [T1	E_2
25	T_1 & T_2	T2	T2 [T1	E_1 & E_2

Appendix E - Test cases for Control Application

In the table below each test case for the control application is presented. The first column contains a unique number for the test case. Columns S_1 to S_6 represent event occurrences in respective observation intervals. Columns E_3 to E_4 represent possible execution states of transactions T_3 and T_4 respectively. Rows which are in **boldface** represent test cases that are not applicable. All execution states where transaction T_4 is preempted are invalid because T_4 has a higher priority that T_3. Rows that are ***highlighted*** represent test cases that are not strictly needed, but should be included to see if the assumption about deadlines being with respective periods holds. For example in row 14 arrival of a request for T_4 is observed in interval S_1 and at the same time T_4 is currently executing. This should not happen if deadlines are within respective periods. If invalid test cases and those where an executing transaction is requested are removed only 81 test cases are left.

No.	S_1	S_2	S_3	S_4	S_5	E_4	E_3	No.	S_1	S_2	S_3	S_4	S_5	E_4	E_4
1	0	0	0	0	0	#	#	***11***	***T4***	***0***	***0***	***0***	***0***	***0***	***#***
2	0	0	0	0	0	0	#	***12***	***T4***	***0***	***0***	***0***	***0***	***1***	***#***
3	0	0	0	0	0	1	#	***13***	***T4***	***0***	***0***	***0***	***0***	***#***	***0***
4	0	0	0	0	0	#	0	***14***	***T4***	***0***	***0***	***0***	***0***	***0***	***0***
5	0	0	0	0	0	0	0	***15***	***T4***	***0***	***0***	***0***	***0***	***1***	***0***
6	0	0	0	0	0	1	0	**16**	**T4**	**0**	**0**	**0**	**0**	**#**	**1**
7	**0**	**0**	**0**	**0**	**0**	**#**	**1**	**17**	**T4**	**0**	**0**	**0**	**0**	**0**	**1**
8	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**18**	**T4**	**0**	**0**	**0**	**0**	**1**	**1**
9	**0**	**0**	**0**	**0**	**0**	**1**	**1**	19	0	T4	0	0	0	#	#
10	T4	0	0	0	0	#	#	***20***	***0***	***T4***	***0***	***0***	***0***	***0***	***#***

21	***0***	***T4***	***0***	***0***	***0***	***1***	***#***	**53**	**0**	**0**	**0**	**0**	**T4**	**0**	**1**
22	0	T4	0	0	0	#	0	**54**	**0**	**0**	**0**	**0**	**T4**	**1**	**1**
23	***0***	***T4***	***0***	***0***	***0***	***0***	***0***	55	T3	0	0	0	0	#	#
24	***0***	***T4***	***0***	***0***	***0***	***1***	***0***	56	T3	0	0	0	0	0	#
25	**0**	**T4**	**0**	**0**	**0**	**#**	**1**	57	**T3**	**0**	**0**	**0**	**0**	**1**	**#**
26	**0**	**T4**	**0**	**0**	**0**	**0**	**1**	*58*	***T3***	***0***	***0***	***0***	***0***	***#***	***0***
27	**0**	**T4**	**0**	**0**	**0**	**1**	**1**	*59*	***T3***	***0***	***0***	***0***	***0***	***0***	***0***
28	0	0	T4	0	0	#	#	*60*	***T3***	***0***	***0***	***0***	***0***	***1***	***0***
29	***0***	***0***	***T4***	***0***	***0***	***0***	***#***	**61**	**T3**	**0**	**0**	**0**	**0**	**#**	**1**
30	***0***	***0***	***T4***	***0***	***0***	***1***	***#***	**62**	**T3**	**0**	**0**	**0**	**0**	**0**	**1**
31	0	0	T4	0	0	#	0	**63**	**T3**	**0**	**0**	**0**	**0**	**1**	**1**
32	***0***	***0***	***T4***	***0***	***0***	***0***	***0***	64	0	T3	0	0	0	#	#
33	***0***	***0***	***T4***	***0***	***0***	***1***	***0***	65	0	T3	0	0	0	0	#
34	**0**	**0**	**T4**	**0**	**0**	**#**	**1**	66	0	T3	0	0	0	1	#
35	**0**	**0**	**T4**	**0**	**0**	**0**	**1**	***67***	***0***	***T3***	***0***	***0***	***0***	***#***	***0***
36	**0**	**0**	**T4**	**0**	**0**	**1**	**1**	***68***	***0***	***T3***	***0***	***0***	***0***	***0***	***0***
37	0	0	0	T4	0	#	#	***69***	***0***	***T3***	***0***	***0***	***0***	***1***	***0***
38	***0***	***0***	***0***	***T4***	***0***	***0***	***#***	**70**	**0**	**T3**	**0**	**0**	**0**	**#**	**1**
39	***0***	***0***	***0***	***T4***	***0***	***1***	***#***	**71**	**0**	**T3**	**0**	**0**	**0**	**0**	**1**
40	0	0	0	T4	0	#	0	**72**	**0**	**T3**	**0**	**0**	**0**	**1**	**1**
41	***0***	***0***	***0***	***T4***	***0***	***0***	***0***	73	0	0	T3	0	0	#	#
42	***0***	***0***	***0***	***T4***	***0***	***1***	***0***	74	0	0	T3	0	0	0	#
43	**0**	**0**	**0**	**T4**	**0**	**#**	**1**	75	0	0	T3	0	0	1	#
44	**0**	**0**	**0**	**T4**	**0**	**0**	**1**	***76***	***0***	***0***	***T3***	***0***	***0***	***#***	***0***
45	**0**	**0**	**0**	**T4**	**0**	**1**	**1**	***77***	***0***	***0***	***T3***	***0***	***0***	***0***	***0***
46	0	0	0	0	T4	#	#	***78***	***0***	***0***	***T3***	***0***	***0***	***1***	***0***
47	0	0	0	0	T4	0	#	**79**	**0**	**0**	**T3**	**0**	**0**	**#**	**1**
48	***0***	***0***	***0***	***0***	***T4***	***1***	***#***	**80**	**0**	**0**	**T3**	**0**	**0**	**0**	**1**
49	0	0	0	0	T4	#	0	**81**	**0**	**0**	**T3**	**0**	**0**	**1**	**1**
50	***0***	***0***	***0***	***0***	***T4***	***0***	***0***	82	0	0	0	T3	0	#	#
51	***0***	***0***	***0***	***0***	***T4***	***1***	***0***	83	0	0	0	T3	0	0	#
52	**0**	**0**	**0**	**0**	**T4**	**#**	**1**	84	0	0	0	T3	0	1	#

85	***0***	***0***	***0***	***T3***	***0***	***#***	***0***	**117**	**T4**	**0**	**T3**	**0**	**0**	**1**	**1**
86	***0***	***0***	***0***	***T3***	***0***	***0***	***0***	118	T4	0	0	T3	0	#	#
87	***0***	***0***	***0***	***T3***	***0***	***1***	***0***	***119***	***T4***	***0***	***0***	***T3***	***0***	***0***	***#***
88	**0**	**0**	**0**	**T3**	**0**	**#**	**1**	***120***	***T4***	***0***	***0***	***T3***	***0***	***1***	***#***
89	**0**	**0**	**0**	**T3**	**0**	**0**	**1**	***121***	***T4***	***0***	***0***	***T3***	***0***	***#***	***0***
90	**0**	**0**	**0**	**T3**	**0**	**1**	**1**	***122***	***T4***	***0***	***0***	***T3***	***0***	***0***	***0***
91	0	0	0	0	T3	#	#	***123***	***T4***	***0***	***0***	***T3***	***0***	***1***	***0***
92	0	0	0	0	T3	0	#	**124**	**T4**	**0**	**0**	**T3**	**0**	**#**	**1**
93	0	0	0	0	T3	1	#	**125**	**T4**	**0**	**0**	**T3**	**0**	**0**	**1**
94	***0***	***0***	***0***	***0***	***T3***	***#***	***0***	**126**	**T4**	**0**	**0**	**T3**	**0**	**1**	**1**
95	***0***	***0***	***0***	***0***	***T3***	***0***	***0***	127	T4	0	0	0	T3	#	#
96	***0***	***0***	***0***	***0***	***T3***	***1***	***0***	***128***	***T4***	***0***	***0***	***0***	***T3***	***0***	***#***
97	**0**	**0**	**0**	**0**	**T3**	**#**	**1**	***129***	***T4***	***0***	***0***	***0***	***T3***	***1***	***#***
98	**0**	**0**	**0**	**0**	**T3**	**0**	**1**	***130***	***T4***	***0***	***0***	***0***	***T3***	***#***	***0***
99	**0**	**0**	**0**	**0**	**T3**	**1**	**1**	***131***	***T4***	***0***	***0***	***0***	***T3***	***0***	***0***
100	T4	T3	0	0	0	#	#	***132***	***T4***	***0***	***0***	***0***	***T3***	***1***	***0***
101	***T4***	***T3***	***0***	***0***	***0***	***0***	***#***	**133**	**T4**	**0**	**0**	**0**	**T3**	**#**	**1**
102	***T4***	***T3***	***0***	***0***	***0***	***1***	***#***	**134**	**T4**	**0**	**0**	**0**	**T3**	**0**	**1**
103	***T4***	***T3***	***0***	***0***	***0***	***#***	***0***	**135**	**T4**	**0**	**0**	**0**	**T3**	**1**	**1**
104	***T4***	***T3***	***0***	***0***	***0***	***0***	***0***	136	0	T4	T3	0	0	#	#
105	***T4***	***T3***	***0***	***0***	***0***	***1***	***0***	***137***	***0***	***T4***	***T3***	***0***	***0***	***0***	***#***
106	**T4**	**T3**	**0**	**0**	**0**	**#**	**1**	***138***	***0***	***T4***	***T3***	***0***	***0***	***1***	***#***
107	**T4**	**T3**	**0**	**0**	**0**	**0**	**1**	***139***	***0***	***T4***	***T3***	***0***	***0***	***#***	***0***
108	**T4**	**T3**	**0**	**0**	**0**	**1**	**1**	***140***	***0***	***T4***	***T3***	***0***	***0***	***0***	***0***
109	T4	0	T3	0	0	#	#	***141***	***0***	***T4***	***T3***	***0***	***0***	***1***	***0***
110	***T4***	***0***	***T3***	***0***	***0***	***0***	***#***	**142**	**0**	**T4**	**T3**	**0**	**0**	**#**	**1**
111	***T4***	***0***	***T3***	***0***	***0***	***1***	***#***	**143**	**0**	**T4**	**T3**	**0**	**0**	**0**	**1**
112	***T4***	***0***	***T3***	***0***	***0***	***#***	***0***	**144**	**0**	**T4**	**T3**	**0**	**0**	**1**	**1**
113	***T4***	***0***	***T3***	***0***	***0***	***0***	***0***	145	0	T4	0	T3	0	#	#
114	***T4***	***0***	***T3***	***0***	***0***	***1***	***0***	***146***	***0***	***T4***	***0***	***T3***	***0***	***0***	***#***
115	**T4**	**0**	**T3**	**0**	**0**	**#**	**1**	***147***	***0***	***T4***	***0***	***T3***	***0***	***1***	***#***
116	**T4**	**0**	**T3**	**0**	**0**	**0**	**1**	***148***	***0***	***T4***	***0***	***T3***	***0***	***#***	***0***

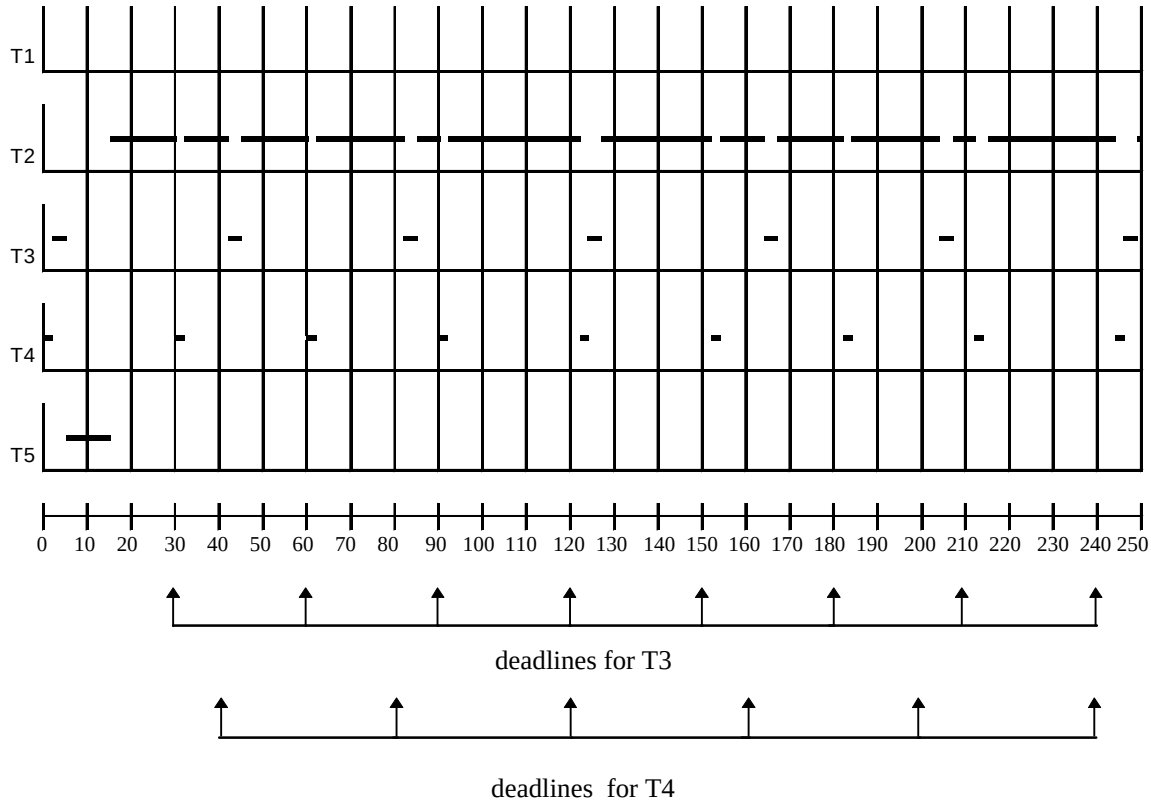
149	***0***	***T4***	***0***	***T3***	***0***	***0***	***0***	181	0	0	0	T4	T3	#	#
150	***0***	***T4***	***0***	***T3***	***0***	***1***	***0***	***182***	***0***	***0***	***0***	***T4***	***T3***	***0***	***#***
151	**0**	**T4**	**0**	**T3**	**0**	**#**	**1**	***183***	***0***	***0***	***0***	***T4***	***T3***	***1***	***#***
152	**0**	**T4**	**0**	**T3**	**0**	**0**	**1**	***184***	***0***	***0***	***0***	***T4***	***T3***	***#***	***0***
153	**0**	**T4**	**0**	**T3**	**0**	**1**	**1**	***185***	***0***	***0***	***0***	***T4***	***T3***	***0***	***0***
154	0	T4	0	0	T3	#	#	***186***	***0***	***0***	***0***	***T4***	***T3***	***1***	***0***
155	***0***	***T4***	***0***	***0***	***T3***	***0***	***#***	**187**	**0**	**0**	**0**	**T4**	**T3**	**#**	**1**
156	***0***	***T4***	***0***	***0***	***T3***	***1***	***#***	**188**	**0**	**0**	**0**	**T4**	**T3**	**0**	**1**
157	***0***	***T4***	***0***	***0***	***T3***	***#***	***0***	**189**	**0**	**0**	**0**	**T4**	**T3**	**1**	**1**
158	***0***	***T4***	***0***	***0***	***T3***	***0***	***0***	190	T3	T4	0	0	0	#	#
159	***0***	***T4***	***0***	***0***	***T3***	***1***	***0***	***191***	***T3***	***T4***	***0***	***0***	***0***	***0***	***#***
160	**0**	**T4**	**0**	**0**	**T3**	**#**	**1**	***192***	***T3***	***T4***	***0***	***0***	***0***	***1***	***#***
161	**0**	**T4**	**0**	**0**	**T3**	**0**	**1**	***193***	***T3***	***T4***	***0***	***0***	***0***	***#***	***0***
162	**0**	**T4**	**0**	**0**	**T3**	**1**	**1**	***194***	***T3***	***T4***	***0***	***0***	***0***	***0***	***0***
163	**0**	**0**	**T4**	**T3**	**0**	**#**	**#**	***195***	***T3***	***T4***	***0***	***0***	***0***	***1***	***0***
164	***0***	***0***	***T4***	***T3***	***0***	***0***	***#***	**196**	**T3**	**T4**	**0**	**0**	**0**	**#**	**1**
165	***0***	***0***	***T4***	***T3***	***0***	***1***	***#***	**197**	**T3**	**T4**	**0**	**0**	**0**	**0**	**1**
166	***0***	***0***	***T4***	***T3***	***0***	***#***	***0***	**198**	**T3**	**T4**	**0**	**0**	**0**	**1**	**1**
167	***0***	***0***	***T4***	***T3***	***0***	***0***	***0***	199	T3	0	T4	0	0	#	#
168	***0***	***0***	***T4***	***T3***	***0***	***1***	***0***	*200*	*T3*	*0*	*T4*	*0*	*0*	*0*	*#*
169	**0**	**0**	**T4**	**T3**	**0**	**#**	**1**	*201*	*T3*	*0*	*T4*	*0*	*0*	*1*	*#*
170	**0**	**0**	**T4**	**T3**	**0**	**0**	**1**	*202*	*T3*	*0*	*T4*	*0*	*0*	*#*	*0*
171	**0**	**0**	**T4**	**T3**	**0**	**1**	**1**	*203*	*T3*	*0*	*T4*	*0*	*0*	*0*	*0*
172	**0**	**0**	**T4**	**0**	**T3**	**#**	**#**	*204*	*T3*	*0*	*T4*	*0*	*0*	*1*	*0*
173	***0***	***0***	***T4***	***0***	***T3***	***0***	***#***	**205**	**T3**	**0**	**T4**	**0**	**0**	**#**	**1**
174	***0***	***0***	***T4***	***0***	***T3***	***1***	***#***	**206**	**T3**	**0**	**T4**	**0**	**0**	**0**	**1**
175	***0***	***0***	***T4***	***0***	***T3***	***#***	***0***	**207**	**T3**	**0**	**T4**	**0**	**0**	**1**	**1**
176	***0***	***0***	***T4***	***0***	***T3***	***0***	***0***	208	T3	0	0	T4	0	#	#
177	***0***	***0***	***T4***	***0***	***T3***	***1***	***0***	***209***	***T3***	***0***	***0***	***T4***	***0***	***0***	***#***
178	**0**	**0**	**T4**	**0**	**T3**	**#**	**1**	***210***	***T3***	***0***	***0***	***T4***	***0***	***1***	***#***
179	**0**	**0**	**T4**	**0**	**T3**	**0**	**1**	***211***	***T3***	***0***	***0***	***T4***	***0***	***#***	***0***
180	**0**	**0**	**T4**	**0**	**T3**	**1**	**1**	***212***	***T3***	***0***	***0***	***T4***	***0***	***0***	***0***

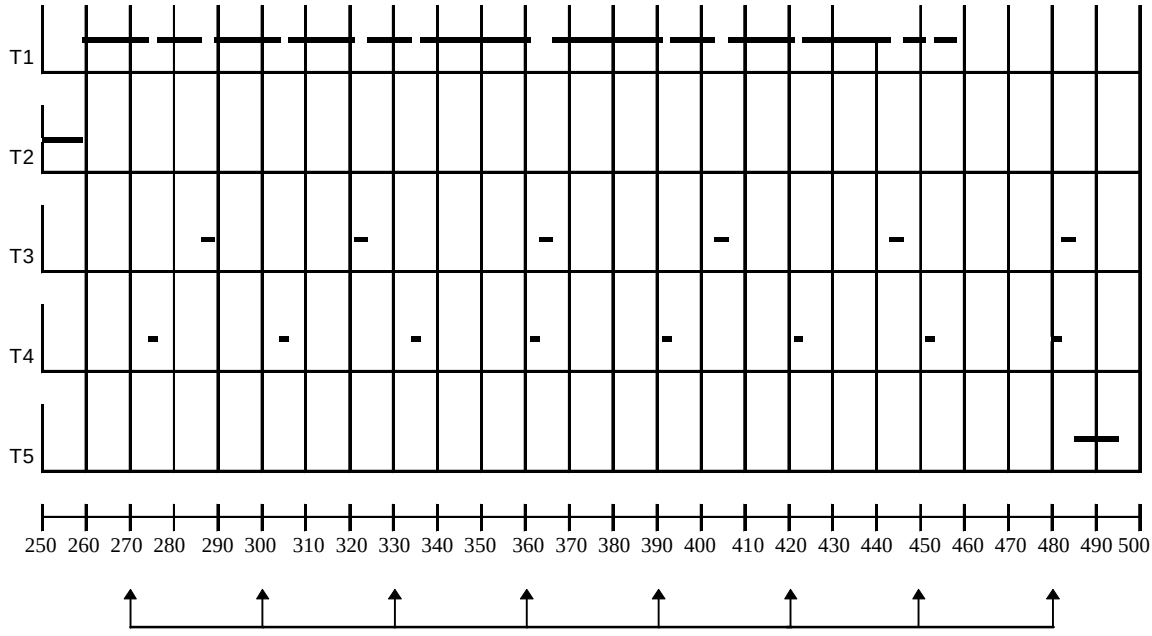
213	***T3***	***0***	***0***	***T4***	***0***	***1***	***0***	***245***	***0***	***T3***	***0***	***0***	***T4***	***0***	***#***
214	**T3**	**0**	**0**	**T4**	**0**	**#**	**1**	***246***	***0***	***T3***	***0***	***0***	***T4***	***1***	***#***
215	**T3**	**0**	**0**	**T4**	**0**	**0**	**1**	***247***	***0***	***T3***	***0***	***0***	***T4***	***#***	***0***
216	**T3**	**0**	**0**	**T4**	**0**	**1**	**1**	***248***	***0***	***T3***	***0***	***0***	***T4***	***0***	***0***
217	**T3**	**0**	**0**	**0**	**T4**	**#**	**#**	***249***	***0***	***T3***	***0***	***0***	***T4***	***1***	***0***
218	***T3***	***0***	***0***	***0***	***T4***	***0***	***#***	**250**	**0**	**T3**	**0**	**0**	**T4**	**#**	**1**
219	***T3***	***0***	***0***	***0***	***T4***	***1***	***#***	**251**	**0**	**T3**	**0**	**0**	**T4**	**0**	**1**
220	***T3***	***0***	***0***	***0***	***T4***	***#***	***0***	**252**	**0**	**T3**	**0**	**0**	**T4**	**1**	**1**
221	***T3***	***0***	***0***	***0***	***T4***	***0***	***0***	253	0	0	T3	T4	0	#	#
222	***T3***	***0***	***0***	***0***	***T4***	***1***	***0***	***254***	***0***	***0***	***T3***	***T4***	***0***	***0***	***#***
223	**T3**	**0**	**0**	**0**	**T4**	**#**	**1**	***255***	***0***	***0***	***T3***	***T4***	***0***	***1***	***#***
224	**T3**	**0**	**0**	**0**	**T4**	**0**	**1**	***256***	***0***	***0***	***T3***	***T4***	***0***	***#***	***0***
225	**T3**	**0**	**0**	**0**	**T4**	**1**	**1**	***257***	***0***	***0***	***T3***	***T4***	***0***	***0***	***0***
226	**0**	**T3**	**T4**	**0**	**0**	**#**	**#**	***258***	***0***	***0***	***T3***	***T4***	***0***	***1***	***0***
227	***0***	***T3***	***T4***	***0***	***0***	***0***	***#***	**259**	**0**	**0**	**T3**	**T4**	**0**	**#**	**1**
228	***0***	***T3***	***T4***	***0***	***0***	***1***	***#***	**260**	**0**	**0**	**T3**	**T4**	**0**	**0**	**1**
229	***0***	***T3***	***T4***	***0***	***0***	***#***	***0***	**261**	**0**	**0**	**T3**	**T4**	**0**	**1**	**1**
230	***0***	***T3***	***T4***	***0***	***0***	***0***	***0***	262	0	0	T3	0	T4	#	#
231	***0***	***T3***	***T4***	***0***	***0***	***1***	***0***	***263***	***0***	***0***	***T3***	***0***	***T4***	***0***	***#***
232	**0**	**T3**	**T4**	**0**	**0**	**#**	**1**	***264***	***0***	***0***	***T3***	***0***	***T4***	***1***	***#***
233	**0**	**T3**	**T4**	**0**	**0**	**0**	**1**	***265***	***0***	***0***	***T3***	***0***	***T4***	***#***	***0***
234	**0**	**T3**	**T4**	**0**	**0**	**1**	**1**	***266***	***0***	***0***	***T3***	***0***	***T4***	***0***	***0***
235	0	T3	0	T4	0	#	#	***267***	***0***	***0***	***T3***	***0***	***T4***	***1***	***0***
236	***0***	***T3***	***0***	***T4***	***0***	***0***	***#***	**268**	**0**	**0**	**T3**	**0**	**T4**	**#**	**1**
237	***0***	***T3***	***0***	***T4***	***0***	***1***	***#***	**269**	**0**	**0**	**T3**	**0**	**T4**	**0**	**1**
238	***0***	***T3***	***0***	***T4***	***0***	***#***	***0***	**270**	**0**	**0**	**T3**	**0**	**T4**	**1**	**1**
239	***0***	***T3***	***0***	***T4***	***0***	***0***	***0***	271	0	0	0	T3	T4	#	#
240	***0***	***T3***	***0***	***T4***	***0***	***1***	***0***	***272***	***0***	***0***	***0***	***T3***	***T4***	***0***	***#***
241	**0**	**T3**	**0**	**T4**	**0**	**#**	**1**	***273***	***0***	***0***	***0***	***T3***	***T4***	***1***	***#***
242	**0**	**T3**	**0**	**T4**	**0**	**0**	**1**	***274***	***0***	***0***	***0***	***T3***	***T4***	***#***	***0***
243	**0**	**T3**	**0**	**T4**	**0**	**1**	**1**	***275***	***0***	***0***	***0***	***T3***	***T4***	***0***	***0***
244	0	T3	0	0	T4	#	#	***276***	***0***	***0***	***0***	***T3***	***T4***	***1***	***0***

277	0	0	0	T3	T4	#	1	309	0	0	0	T4&T3	0	1	#
278	0	0	0	T3	T4	0	1	311	0	0	0	T4&T3	0	#	0
279	0	0	0	T3	T4	1	1	311	0	0	0	T4&T3	0	0	0
280	T4&T3	0	0	0	0	#	#	312	0	0	0	T4&T3	0	1	0
281	T4&T3	0	0	0	0	0	#	313	0	0	0	T4&T3	0	#	1
282	T4&T3	0	0	0	0	1	#	314	0	0	0	T4&T3	0	0	1
283	T4&T3	0	0	0	0	#	0	315	0	0	0	T4&T3	0	1	1
284	T4&T3	0	0	0	0	0	0	316	0	0	0	0	T4&T3	#	#
285	T4&T3	0	0	0	0	1	0	317	0	0	0	0	T4&T3	0	#
286	T4&T3	0	0	0	0	#	1	318	0	0	0	0	T4&T3	1	#
287	T4&T3	0	0	0	0	0	1	319	0	0	0	0	T4&T3	#	0
288	T4&T3	0	0	0	0	1	1	320	0	0	0	0	T4&T3	0	0
289	0	T4&T3	0	0	0	#	#	321	0	0	0	0	T4&T3	1	0
290	0	T4&T3	0	0	0	0	#	322	0	0	0	0	T4&T3	#	1
291	0	T4&T3	0	0	0	1	#	323	0	0	0	0	T4&T3	0	1
292	0	T4&T3	0	0	0	#	0	324	0	0	0	0	T4&T3	1	1
293	0	T4&T3	0	0	0	0	0								
294	0	T4&T3	0	0	0	1	0								
295	0	T4&T3	0	0	0	#	1								
296	0	T4&T3	0	0	0	0	1								
297	0	T4&T3	0	0	0	1	1								
298	0	0	T4&T3	0	0	#	#								
299	0	0	T4&T3	0	0	0	#								
300	0	0	T4&T3	0	0	1	#								
301	0	0	T4&T3	0	0	#	0								
302	0	0	T4&T3	0	0	0	0								
303	0	0	T4&T3	0	0	1	0								
304	0	0	T4&T3	0	0	#	1								
305	0	0	T4&T3	0	0	0	1								
306	0	0	T4&T3	0	0	1	1								
307	0	0	0	T4&T3	0	#	#								
308	0	0	0	T4&T3	0	0	#								

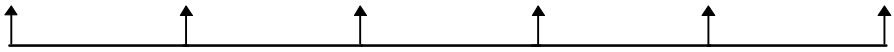
Appendix F - Schedulability of Control Application

Below timelines for the transactions in table 6.1 are presented. The scheduling algorithm used in this timelines is rate-monotonic. In this example a critical instance occurs each 950 ms. The timelines are drawn from 0 to 500 ms. It is sufficient to examine only this range because at 500 ms transactions T4, T3 and T4 have finished and it can be seen that T3 and T3 can not miss their deadlines until the next critical instance at 950 ms. Only deadlines for T4 and T3 are marked. T4 and T3 only have one deadline at 950 ms. During this interval T3 has deadlines at 480 and 960 ms. All deadlines are met.





deadlines for T3



deadlines for T4

Appendix G - Example of algorithm 1

The basic idea is to first create chains of length one followed by chains of length two and so on. The algorithm creates all successive chains and checks if they are valid. For example given a chain {5,4,3,3,2,2} the next largest chain is {5,4,3,3,3,2}. By incrementing the first occurrence of the smallest element in the chain and resetting the rest to the initial value of 2 we always get a chain that is one larger than the original one. To illustrate more specifically how this algorithm works consider finding all possible chains given seven transactions and two resources.

During each iteration an array of length i (loop counter) is assigned to the smallest possible chain {2}, {2,2}, {2,2,2}, and so on. Starting with an initial chain the while loop at line 5 creates successively larger chains. For chains of length one this simply consists of increasing the element by one. In our example during the first iteration through i this would mean that starting from {2} we get the following chains. {2}, {3}, {4}, {5}, {6}, {7}. During the next (and last) iteration of i we start with the smallest chain of length two {2,2}. As before the while loop creates successively larger chains until their sum exceeds that allowed maximum. Increasing the first occurrence of the smallest element in {2,2} results in {3,2} which is a valid chain since 3+2 is less than seven. Increasing the first occurrence of the smallest element in {3,2} results in {3,3} which is a valid chain. From {3,3} we get {4,2} from increasing the first occurrence of 3 and setting the second occurrence back to 2. From {4,2} we get {4,3}which is a valid chain, but also the largest chain of length two given seven transactions. The algorithm terminates with L={ {2}, {3}, {4}, {5}, {6}, {7}, {2,2}, {3,2}, {3,3}, {4,2}, {4,3} }.