Real-Time Test Oracles using Event Monitoring
An Initial Study in Solicitor

Sebastian Nilsson Holmgren
Real-Time Test Oracles using Event Monitoring
An Initial Study in Solicitor

Submitted by Sebastian Nilsson Holmgren to the University of Skövde, as a dissertation for the degree of Bachelor of Science (B.Sc.) at the School of Humanities and Informatics.

June 6, 2005

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

Signed: ____________________________________________

Supervisor: Dr. Jonas Mellin
Real-Time Test Oracles using Event Monitoring
An Initial Study in Solicitor
Sebastian Nilsson Holmgren

Abstract

To gain confidence in that a dynamic real-time system behaves correctly, we test it. Automated verification & validation can be used to conduct testing of such systems in an effective and economic way.

An event monitor can be used as a part of a test oracle to monitor the system that is being tested. The test oracle could use the data (i.e., the streams of events) derived from the tested system, to determine if an executed test case gave a positive or negative result. To do this, the test oracle compares the streams of events received from the event monitor with the event expressions derived from the formal specification, and decides if the executed test case has responded positive or negative. Any deviations between observed behaviour and accepted behaviour should be reported by the test oracle as a negative result. If the executed test case gave a negative result, the monitor part should signal this to the reporter part of the test oracle.

This work aims to investigate how the event expressions can be derived from the formal specification, and in particular, how the event specification language Solicitor can be used to represent these event expressions.

We also discuss the need for parameterized event types in Solicitor, and any other event specification languages used in event monitoring. We also show that support for parameterized event types is a significant requirement for such languages.

Keywords: Dynamic Real-Time Systems, Event Monitoring, Test Oracles, Automated Testing, Solicitor, Event Expressions
Acknowledgements

I would like to thank my supervisor, Dr. Jonas Mellin for all his guidance and helpful tips (both about my project and about \LaTeX) that he has provided me with during my final year project.

I would also like thank Robert Nilsson for giving me an insight into his research, and for providing me with some of the data produced in his research.

I would also like to thank Annika Jakobsen for helping me to proofread my report. Any misspelt words or grammatical errors are entirely my own.

Last but certainly not least, I would like to thank my fiancée, Karin Jakobsen for her support during this project, and for reminding me to eat once in a while.
List of Figures

1  Example of two scheduled tasks .......................... 5
2  Event Monitor Architecture (Mellin 2004, p. 13 fig. 2.1) .... 9
3  Architecture for a Test Environment using an Event Monitor, based on Peters & Parnas (2002, fig. 6) .................. 10
4  From a TAT formal specification to event monitor input, based on personal communication with Robert Nilsson (2005) .... 13
5  Project Focus ........................................... 18
6  Overview of Generation of Code and Header Files ............ 18
7  Client Layers of The Event Monitoring Architecture (Mellin 2004) .................................................. 19
8  Overview of Generation of Executables ........................ 19
9  Execution Window ........................................ 22
10 Precedence Relation - Case 1 ................................ 23
11 Precedence Relation - Case 1, Incorrect Behaviour .......... 23
12 Precedence Relation - Monitoring Interval ................... 23
13 Precedence Relation - Case 2 ................................ 24
14 Precedence Relation - Case 3 ................................ 25
15 Precedence Relation - Case 4 ................................ 25
16 Overlapping Task - Example 1 .............................. 27
17 Event Composition in Recent Context — Without Parameterized Event Types .................................. 28
18 Event Composition in Chronicle Context — With Parameterized Event Types ................................ 29
19 Overlapping Task - Example 2 .............................. 29
20 Event Composition in Chronicle Context — Without Parameterized Event Types ............................... 30
21 Overlapping Durations ...................................... 30
22 Precedence Constraints - Overlapping Tasks ............... 32
1 Introduction

Real-time systems are growing both in number, complexity, size and importance. Since real-time systems are also needed to be more dependable, we need to be sure that we can put our trust in these systems.

One way of gaining trust in that a real-time system behaves correctly is to test it, and then investigate whether the test gave a positive or negative result. Since real-time systems are becoming more complex and manual testing is costly, research has been made in the area of automated testing of these kinds of systems (Chodrow, Jahanian & Donner 1991, Jahanian, Rajkumar & Raju 1994, Nilsson, Offutt & Andler 2004, Nilsson 2003, Peters & Parnas 2002, Peters 2000, Savor & Seviora 1998, Wegener & Mueller 2001).

Automated testing typically uses a formal specification that has been reviewed by domain experts (Peters 2000). From the formal specifications, we can derive a model of correct behaviour for the system that is being tested. This model can then be used by an event monitor and an automatic test oracle to conduct automated testing.

An event monitor can be used as a part of a test oracle to monitor the system that is being tested. The test oracle could use the data (the streams of events) derived from the tested system, to determine if an executed test case gave a positive or negative result. To do this, the test oracle compares the streams of events received from the event monitor with the event expressions derived from the formal specification, and decides if the executed test case has responded positive or negative. Any deviations between observed behaviour and accepted behaviour should be reported by the test oracle as a negative result. If the executed test case gave a negative result, the monitor part should signal this to the reporter part of the test oracle.

In this final year project, we aim to (i) define an architecture for test oracle management, and (ii) investigate how we can express the constraints stated in the formal specification, in the event specification language Solicitor. We will also show that support for parameterized event types is a significant constraint for event specification languages.

In chapter 2, the concepts and terminology of interest for this project are presented. In particular, test oracles, testing of dynamic real-time systems and event monitoring are discussed. Furthermore, the event specification language Solicitor and some of its operators are introduced.

In chapter 3, the problem statement for this project is defined and motivated.

In chapter 4, the method chosen for this project is presented. This includes the research approach, the overview of the solution and the approaches taken to reach the objectives that are presented in chapter 3.
In chapter 5, the results are presented and in chapter 6, the results are analyzed and discussed.

Chapter 7 presents related work of this project.
Finally, chapter 8 summarizes the project, the contribution that this project has made, and gives suggestions for future work.

Symbols, conventions and figure legends are presented in appendix A. Figure legends are presented in appendix B.
2 Background

This chapter provides background to the area of test oracles, automated verification & validation and event monitoring.

When we test a dynamic real-time system, the amount of test cases that must be executed makes it impractical to test the system manually. The reason for this is that we must take notice to at which time events occur, and that the system is dynamically scheduled and event-triggered. Because of this, automated testing has been proposed for these kinds of systems (Nilsson 2003) and much research has been done in the area of automated testing (Chodrow et al. 1991, Jahanian et al. 1994, Nilsson et al. 2004, Nilsson 2003, Peters & Parnas 2002, Peters 2000, Savor & Seviora 1998, Wegener & Mueller 2001).

In this work we aim to investigate how event monitoring can be used as a part of an automated test oracle. The automated test oracle can then be used to automatically determine if a test case has given a positive or negative result. To perform automated testing, an automated test oracle is necessary.

2.1 Test Oracles for Evaluation of Test Results

Test oracles can be divided into two parts, (i) a part that detects erroneous behaviour in the target system, and (ii) a part that reports erroneous behaviour.

This work aims to investigate the use of event monitoring as a part of the detection functionality of automated test oracles. A part of the process of testing is to determine if a system has executed according to the system specification (correct behaviour) or if the execution deviates from the specification (incorrect behaviour). This part of the process can be performed by a test oracle (Peters & Parnas 2002).

Often, the test oracle is the person that is testing. That is, the person executes the test case and then manually determines if the test case revealed any fault by reviewing the result. In this report, the term test oracle refers to a program that automatically compares the results of the execution of test cases with a formal specification of the system and determines whether or not the behaviour of the execution deviates from the specification.

2.2 Automated Verification & Validation

In this work, we make no distinction between the terms verification & validation, in contrast to Boehm (1981, p. 37).
2.2 Automated Verification & Validation

We emphasize dynamic verification techniques, that, according to Laprie (1994), imply that the system is executed during verification. These techniques are referred to as testing. We do not emphasize the other alternative, static verification techniques, that do not imply executing the system; the reason is that our work emphasizes test oracles, a feature that is not required for static verification techniques.

Automated verification and validation typically use the formal specification of a system (Mellin 2004, Peters 2000, Peters & Parnas 2002, Savor & Seviola 1998). The behaviour of the target system is monitored and compared with the system’s specification. This can be done in several ways, for example, (Peters & Parnas 2002, Jahanian et al. 1994, Mellin 2004).

According to Nilsson (2003), the timeliness of the system is affected by the order in which tasks execute and by hardware caches. This implies that testing such systems becomes harder since more factors influence the behaviour during testing. For example, it is harder to perform regression testing\(^1\) on such a system since interrupts may have affected the execution order of different tasks in the system. This means that manual testing of such a system becomes very time-consuming and expensive.

Since non-automated verification becomes time-consuming and expensive, automated verification is required. Additionally, automated verification simplifies regression testing. To perform automated verification, three different parts need to be automated:

- Test case generation
- Test case execution
- Test case execution result analysis

We conjecture that it is less time-consuming to automate test case generation and test case execution then to manually generate and execute them. Automated test case generation and test case execution are thus needed because we want all steps in the process testing to be automated so that we do not have any bottlenecks.

Automatic test case execution results analysis is necessary to be able to determine how the target system behaved during testing. This step should also be automated, since we do not want to go through all the test case execution data.

\(^1\)According to Laprie (1994), testing that is performed after one or several corrections has been made to the software, after it has first been tested, is termed regression testing. Regression testing serves to make sure that the corrections made to the software has removed the fault they were meant to, and that no new faults have been introduced in the system by the corrections.
executions by hand. Event monitoring (analysis of streams of events on the target system) can be useful when executing test cases, and to perform automatic analysis of test cases, a test oracle is required. We use an example to show how an event monitor can be used as a test oracle.

Example 1: Using an Event Monitor as a part of a Test Oracle
Assume we have two tasks, task A and task B (fig. 1). We have two constraints: (i) the execution of task A must precede the execution of task B; (ii) the deadline of task A precedes that of task B.

Assume further that an event $A_{\text{start}}$ occurs when task A starts to execute and that $A_{\text{term}}$ occurs when task A is terminated. Similarly, $B_{\text{start}}$ and $B_{\text{term}}$ occurs when task B is started and terminated.

To illustrate how these events can be used to express sequences of events, the event specification language Solicitor’s sequence operator (Mellin 2004), will be briefly introduced. The expression: $E = E' ; E''$, means that a composite event expression of type $E$ occurs when an event occurrence $E'$ is followed by an occurrence of event $E''$.

Given the aforementioned situation, we want to be able to tell if the execution of these two tasks has been done correctly or not. The correct behaviour (of the execution of the two tasks) can be represented by the composite event expression:

- $E_{\text{correct}} = A_{\text{start}}; A_{\text{term}}; B_{\text{start}}; B_{\text{term}}$

Any other composite event expression represents a behaviour which is incorrect. For example the following permutations of the correct composite event expression:

- $E_{\text{incorrect}} = A_{\text{start}}; B_{\text{start}}; A_{\text{term}}; B_{\text{term}}$

- $E_{\text{incorrect'}} = B_{\text{start}}; B_{\text{term}}; A_{\text{start}}; B_{\text{term}}$
Or the following composite event expression, which is not a permutation of the correct composite event expression (the letter $N$ denotes the Solicitor non-occurrence operator. See definition 3 p. 12):

- $E_{\text{incorrect}'} = N(System\text{Start}, A_{\text{start}}, B_{\text{start}})$

These behaviours should be reported by the test oracle as unacceptable. An event monitor should monitor the system and analyze the events that occur in the system. Based on the events, the event monitor should determine if the behaviour of the system is correct (e.g., $E_{\text{correct}}$) or incorrect (e.g., $E_{\text{incorrect}}, E_{\text{incorrect}'}, E_{\text{incorrect}''}$ or any other incorrect behaviour). If the event monitor fulfils these tasks, it could also be used as a part of a test oracle.

### 2.2.1 Formal System Specification

A formal system specification specifies the correct behaviour of a system in a precise and unambiguous way. Since the formal specification is precise and unambiguous, we can use it to generate the test oracle automatically, one part of this is event monitoring.

### 2.3 Testing

To quote Pressman: “Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design and code generation” (Pressman 2000, p. 426). Further, according to Schütz (1994): “testing is one of the most widely known and most widely applied verification techniques, and is thus of large practical importance”.

We test software to increase the quality of that software (i.e., we try to reveal and remove as many faults as possible). The idea is “the fewer the faults the higher the quality”. When building dependable software, it is imperative that the software is thoroughly tested, so that the software can be trusted.

### 2.3.1 Observability

According to Schütz (1994), we must observe the system that is being tested, so that we can evaluate the correctness of the system behaviour. For this to be possible, the system must be observable. We can say that the observability of the system denotes how easy it is to observe the system.
Whenever we observe something, we affect it (the Heisenberg principle (Heisenberg 1989)). In the context of monitoring of software (to achieve observability for a real-time system), we might typically insert some code in the monitored system, that we can use to reveal the inner state of the system. This code will affect the monitored system (e.g., execution time). According to Mellin (2004, based on (Gait 1985)), the *probe effect* is the difference in behaviour between a monitored object and its unmonitored counterpart.

When we are monitoring real-time systems, the probe effect is a major issue. If we monitor a real-time system, it might work fine (i.e., it meets its deadlines) during testing. However, if we remove the monitoring functionality and deploy the system in its operational environment, it might miss its deadlines. This could happen because the monitoring functionality affected the order in which the tasks were executed and when the functionality was removed, the behaviour of the system changed.

Schütz (1994) states that the probe-effect can be avoided. Further, he states, that one way of accomplishing this is by leaving all monitoring and testing support in the system during its productive operation.

### 2.3.2 Dynamic Real-Time Systems

There are different types of real-time systems. According to Kopetz & Verissimo (1993), there are two main types of real-time systems (or approaches to the design of real-time systems): *event-triggered* systems and *time-triggered* systems. An event-triggered system reacts to significant external events directly and immediately, while a time-triggered system reacts to significant external events at pre-specified instants.

According to Kopetz (1993) real-time systems can also be split up into *statically scheduled* systems and *dynamically scheduled* systems. When a system is statically scheduled, all the tasks that will execute on the system are known in advance. The information about the tasks and the task-load is used to create a schedule before the system is executed. In contrast, a dynamically scheduled system changes its schedule over time, depending on the task-load and the environment that the system operates in.

This work emphasizes testing of *dynamic real-time systems*. A dynamic real-time system is a system that is dynamically scheduled and event-triggered (Nilsson 2003).

### 2.3.3 Issues in Testing of Real-Time Systems

Testing real-time systems differs from testing non-real-time systems in several aspects. According to Schütz (1994), there are six fundamental issues when
testing distributed real-time systems. Since this work emphasizes automated verification & validation of real-time systems, three of those are of special interest in this work:

- Reproducibility
- Observability
- Representativity

Reproducibility is important since we want to be able to conduct automated regression testing. Observability is important since we have to be able to observe the system that we want to monitor. Finally, representativity is important since we want to test the system in a realistic way (e.g., with realistic input).

Further, according to Schütz (1994), there are three different types of events that are necessary to consider to be able to conduct testing of real-time systems. In this work, we emphasize one of them: synchronization-point events. We do not emphasize the other two: interrupts and access to time. The reason for this is that, if we can not conduct automated testing without interrupts or access to time, we cannot do it with them.

The factors presented by Nilsson (2003) in section 2.2, we can seldom (or never) control during testing, or during real operation of the system. Due to this non-determinism, we must execute the same test case several times and use statistical methods, so that we can be sure that the system will not fail more often then what is acceptable (Nilsson 2003, p. 11). For example, we might execute a certain test case 1000 times, and then use statistical methods to show that for 99% of the test case executions, the system has not failed.

2.4 Event Monitoring for Testing

This work emphasizes event monitoring. We will, however, start by introducing monitoring, and give some definitions of a monitor. Then we will introduce event monitoring and give a definition of an event monitor.

2.4.1 Monitoring

According to Peters & Parnas (2002): “A monitor is a system that observes the behaviour of a system and determines if it is consistent with a given specification.” Two other important terms are defined by Peters (2000): The target system and the monitor system. The target system is the system
to be monitored, and the monitor system is the system that observes the behaviour of the target system.

Savor & Seviola (1998) defines the term *software supervisor*: “A software supervisor observes the inputs and outputs of a target system. It uses a model of correct behavior, derived from the target system’s requirements specification.”

### 2.4.2 Event Monitoring

In this work, we will adopt the definition of an event monitor as it was defined by Mellin (2004, p. 11): “an event monitor is a construct for the monitoring of events (or properties) of a system; this brief definition follows the definition of Snodgrass (1988): monitoring is the extraction of dynamic information concerning the (computational) process.”

Figure 2 gives an overview of a possible architecture of an event monitor. For more information about monitoring and event monitoring see (Peters 2000, Peters & Parnas 2002, Mellin 2004).

![Event Monitor Architecture](image_url)

**Figure 2: Event Monitor Architecture (Mellin 2004, p. 13 fig. 2.1)**

Event monitoring can be used in automated testing. In figure 3, an example architecture of a test environment that uses an event monitor is depicted.

When using an event monitor to perform automated testing, some kind of input is needed. In figure 3, the leftmost arrow symbolizes this input. The input (e.g., test cases) is generated automatically from formal specification for the system that is being tested.

Peters & Parnas (2002) have presented a method for monitoring a test-object. In their method, the input is fed into the executing test object and the event monitor. When the test object processes the input, events
2.4 Event Monitoring for Testing

are generated and these are signalled to the event monitor. The event monitor compares the output given by the test object (the signalled events) with the expected result of the matching input. The expected results are generated/calculated from the same formal specification as the test cases. The monitor then gives the data to a test oracle\(^2\) that determines whether or not the test object has behaved according to the formal specification. If it has not, the event monitor takes some kind of action.

2.4.3 The Solicitor Event Specification Language

When we are conducting event monitoring, we need some way of describing events (i.e., primitive events) and patterns of events (i.e., composite events). According to Mellin (2004, p. 40, based on (Chakravarthy, Krishnaprasad, Anwar & Kim 1994)), “primitive event types are pre-defined events in the monitored object denoting that something important has occurred”. Further, “composite event types are defined by an event expression where an event expression is a pattern that defines how event occurrences of defined types can be combined”.

We use the event specification language Solicitor (Mellin 2004) to represent primitive and composite event expressions. In this work, we adopt the following notation: \(E\) is an event; \(t\) denotes a time; \([t, t']\) denotes an interval between \(t\) and \(t'\) such that \(t \leq t'\); \(T\) denotes a duration; \(\lor\) denotes the boolean OR operator; and finally, \(\text{now}\) denotes the current time.

The predicate \(x \overset{\Delta}{=} y\) is used to define predicates in terms of other predicates, where \(\forall\) unbound variables that appear both in \(P\) and \(Q\) \((P \iff Q) \iff P \overset{\Delta}{=} Q\), and the \(\text{occ}(E, [t, t'])\) predicate is true if an occurrence of \(E\)

\(^2\)As described earlier in example 1, the event monitor and the test oracle might be the same software component.
that started at time $t$ and terminated at time $t'$ has occurred (Mellin 2004).

To illustrate the kind of event expressions that we might encounter during the practical part of the project, a small example will be used.

**Example 2: Important event expressions**

Assume that we have two processes, $P_1$ and $P_2$, that communicate using Remote Procedure Call (RPC), and that we want to monitor the communication between the two processes. We have a bound on the communication time, that states that the transmission for a message $m$, sent (via RPC) from processes $P_1$ to process $P_2$, can take maximum of $T_{\text{max}}$ time. Further, since RPC is a call and reply (return) architecture, if $P_1$ sends a message to $P_2$, $P_2$ must send a response to $P_1$, before $P_1$ can send another message.

If we want to model this scenario by using an event specification language, we need some operators. For example: (i) we want to be able to specify timeouts, so that we can monitor if the bound on the communication time is violated; (ii) we also want to be able to specify in which order things will happen (e.g., $P_1$ sends a message to $P_2$); (iii) we want to be able to specify events or event patterns that have not occurred (i.e., non-occurrence of a certain event pattern) and, finally, (iv) we want to be able to specify the inverse of non-occurrence (i.e., if non-occurrence is used to express correct behaviour, we want to be able to express incorrect behaviour).

The three operators are each defined in first-order predicate logic (Mellin 2004, pp. 47–49, based on (Galton & Augusto 2001)):

- **Definition 1 Relative temporal:**

  $\text{occ}(E_1+T;[t,t']) \triangleq \exists t_1(\text{occ}(E_1,[t,t_1]) \land t'=t+T \land t' \leq \text{now})$

  This predicate is used to specify timeouts. The predicate is true if an event of type $E$ has started at time $t$ and terminated at time $t'$ and time $t'$ is equal to $t + T$ and $t'$ is less or equal of the current time.
• **Definition 2 Sequence:**

\[
\text{occ}(E_1;E_2,[t,t']) \triangleq \exists t_1 < t_2 (\text{occ}(E_1,[t,t_1]) \land \text{occ}(E_2,[t_2,t']))
\]

This predicate is used to express sequences of events. The predicate is true if event \(E_1\) occurs before event \(E_2\).

• **Definition 3 Non-occurrence:**

\[
\text{occ}(N(E_1,E_2,E_3),[t,t']) \triangleq \\
\exists t_1 < t_2 (\text{occ}(E_1,[t,t_1]) \land \text{occ}(E_3,[t_2,t'])) \land \\
\forall t_3 \leq t_4 (t_1 \leq t_3 \land t_4 \leq t_2 \Rightarrow \neg \text{occ}(E_2,[t_3,t_4]))
\]

This predicate is used to express non-occurrences of events. The predicate is true if no event \(E_2\) occurs in an interval opened by an \(E_1\) event and closed by an \(E_3\) event.

• **Definition 4 Aperiodic:**

\[
\text{occ}(A(E_1,E_2,E_3),[t,t']) \triangleq \\
\exists t_1 \leq t_2 (\text{occ}(E_1,[t_1,t_2]) \land \text{occ}(E_2,[t,t'])) \land t \geq t_2 \land \\
\forall t_3 \leq t_4 (t_2 \leq t_3 \land t_4 \leq t' \Rightarrow \neg \text{occ}(E_3,[t_3,t_4]))
\]

This predicate is true if an \(E_2\) event occurs in an interval opened by an \(E_1\) event and closed by an \(E_3\) event (i.e., it is the inverse of the non-occurrence operator).

**Example 3: Message transmission**

We can use the sequence and non-occurrence operators to express the correct and an incorrect behavior for the system presented in example 2.

An omission failure can be caught by the following expression:

• \(E_{\text{omission}} = N(\text{call}, \text{reply}, \text{call} + 2T)\)

A possible correct behaviour can be expressed as:

• \(E_{\text{correct}} = \text{call};(\text{expire}: 2T)\text{reply}\)

where \(\text{expire}\) means that the reply event expires within 2\(T\) from the call event.

\[\square\]
2.4.4 Event Contexts

According to Mellin (2004), Buchman (1994) states that, support for recent and chronicle contexts is a minimum requirement for real-time systems. Hence, these two event contexts are of interest in this work.

- Recent Context
  According to Mellin (2004, p. 57, based on, (Chakravarthy et al. 1994)):
  “In this context, only the most recent occurrences are allowed to contribute to composite events”.

- Chronicle Context
  According to Mellin (2004, p. 57, based on, (Chakravarthy et al. 1994)):
  “In this context, only occurrences with matching order of occurrence are allowed to contribute to composite events”.

2.5 Automation of Timeliness Testing

In section 2.2, three core concepts that are required by automated verification are presented. Nilsson et al. (2004) have presented a method that can be used to realize automated test case generation & execution. This method is illustrated in figure 4.

![Diagram](image.png)

Figure 4: From a TAT formal specification to event monitor input, based on personal communication with Robert Nilsson (2005)
Nilsson’s method utilize a formal specification over a system as input, and automatically generates test cases that can be executed. In this method the formal specifications are modelled as TAT specifications (Fersman, Petterson & Yi 2002). The formal specifications are fed into a mutant generator. The mutant generator uses a set of mutation primitives to mutate the TAT specification. This is done to reveal as many design faults as possible. The mutation primitives vary the data that has been estimated during the design of the system.

The mutated specifications are then fed into the specification analyzer that analyzes\(^3\) them. Any mutation that is likely to cause an error is marked as killed. The other mutations are discarded. The killed mutants are stored/logged in a trace. From the data in the trace, test cases are generated.

---

\(^3\)Currently the specification is analyzed by a model-checker. Ongoing work aims to replace the model-checker with heuristics to improve scalability (Robert Nilsson, personal communication, 2005).
3 Problem Definition

The purpose of this project is to investigate the use of event monitoring as a part of automated test oracles. That is, how to use event monitoring for detection of erroneous behaviours and states in the test object.

3.1 Motivation

There is a need for efficient analysis of event traces to perform detection. To be able to conduct automated testing, which is highly desirable according to Nilsson (2003), an automatic test oracle is necessary. Without a test oracle, we are not able to automatically distinguish between correct and incorrect behaviours.

As described in section 2.3.3, testing of real-time systems is more difficult and problematic than testing non-real-time systems. Event monitoring and automated test oracles can be used to make the testing less problematic and more accurate. Since we cannot only analyze how the system will behave in advance (we can in some situations), we need to monitor the system runtime. By using event monitoring and automated test oracles, we can monitor the system when it is executing in a test environment, to see if it behaves acceptably (according to its specification). This is illustrated in figure 3.

Event monitoring is an existing feature and much research has been done in the area of event monitoring (e.g., (Chakravarthy, Blaustein, Buchmann, Carey, Dayal, Goldhirsh, Hsu, Jauhuri, Ladin, Livney, McCarthy, McKee & Rosenthal 1989, Chodrow et al. 1991, Jahanian et al. 1994, Mellin 2004, Peters 2000, Peters & Parnas 2002)).

A benefit of using event monitoring for automated test oracles is that we can avoid the probe effect (Mellin 2004, p. 15) (see section 2.3.1) when testing, since the feature (i.e., event monitoring) is already in the system. The reason for this is, that no extra functionally would be needed in the tested system, only in the event monitor. This corresponds to one of Schütz (1994) avoidance strategies.

Additional benefits are: (i) no overload if the expressions are used in the application, and finally, (ii) that we can use the event monitor and the test oracle for on-line diagnostics. That is, we would have means of supervising the system, while it is executing in its real environment. This could lead to a more dependable and reliable system.
3.2 Prerequisites

In section 2.3.2, the concept of dynamic real-time systems are introduced. These kinds of systems are not sufficiently deterministic, for example, we can only partially derive the execution order of tasks and the task load.

For the test oracle to be of any use, we need to have some kind of formal specification that the test oracle can compare the actual behaviour of the system to. Nilsson et al. (2004) have presented a method that generates traces for correct and incorrect behaviour. These traces together with the formal system specification (that is described in section 2.2.1) can be used as input for the event monitor.

3.3 Objectives

To achieve the purpose of this project, these objectives must be fulfilled:

3.3.1 Provide an Overall Architecture for Test Oracle Management

This objective aims to establish an architecture that can be used to automatically generate test oracles for dynamic real-time systems from a formal specification.

3.3.2 Expressing Necessary Constraints in Solicitor

This objective aims to investigate, if it is possible to express the necessary constraints in Solicitor.

If we want to monitor the behaviour of a real-time system, we need some way to express the constraints that are modelled in the systems specification. In this work, we use the monitor architecture (Mellin 2004, ch. 10) and prototype (Mellin 2004, ch. 12) (that has been extensively tested) developed by Jonas Mellin. Using this monitor architecture requires, that the constraints in the formal specification are expressed in the event specification language Solicitor (which is briefly introduced in section 2.4.3).
4 Method

In this section, the approach being used to solve the problems, as described in section 3, is presented.

4.1 Research Approach

When conducting a project in the area of computer science, there are several different approaches that we can choose from. Some of these are literature survey, simulation, experiments, analysis and implementation.

In this project, we have chosen to combine a literature survey with analysis. The literature survey is done to gather background information about the area of automated testing, event monitoring and test oracles. The literature survey will also be used to define an architecture for test oracle management (the first objective).

Analysis will be carried out to fulfil the second objective (defining an overall architecture for test oracle management). For this objective, analysis is suitable since we have a specific problem, to which we do not know if there exists a solution. In this context, analysis means that we will take a certain constraint, and then we will analyze how it can be expressed in Solicitor. The expressions will also be analyzed so that we know what behaviours the expression covers. Examples of constraints that will be analyzed are time constraints, precedence constraints and shared resource constraints.

As stated above, in this project a combination of a literature survey and analysis is the approach of choice. The reason for this is that these approaches are suitable for our prerequisites. Simulation is not suitable since we yet do not know if all significant constraints can be expressed. Therefore, it is better to analyze this first. Experiments is not suitable since we do not have an implemented test oracle to conduct experiments on. Implementation will not be conducted since this is an initial study of Solicitor, and not enough knowledge of how Solicitors operators could be used in an automated test oracle is available.

4.2 Overview of Solution: An Architecture for Automated Verification & Validation

Figure 5 illustrates how the system specification (for the target system) is transformed into a monitor specification and configuration of the test oracle reporter.

We start by describing the framework, that is used, when conducting testing with the event monitoring architecture implemented by Jonas Mellin.
4.2 Overview of Solution: An Architecture for Automated Verification & Validation

Figures 6 and 8 together with table 1 illustrates this framework (Jonas Mellin, personal communication, 2005). Figure 7 illustrates the layers of the client side of the event monitoring architecture.

Figure 6 illustrates how the monitor specification is transformed into source and header files for the event monitor:

The event monitor specification is implemented in Solicitor; that is, the formal specification of the target system is translated into Solicitor. This specification is then fed into the Solicitor compiler, that, in the current implementation, transforms it to source code for an executable event monitor. This source code consists of a C++ header file (suffixed .h) and a C++ implementation file. The header file is used to define types and to provide access methods to the event occurrences.
4.3 Expressing Constraints in Solicitor

To be able to use the event monitor as a test oracle, we must be able to specify correct and incorrect behaviours. To do this, the necessary constraints, such as deadlines, precedence relations or periods, must be considered.

To determine whether or not these factors can be expressed in Solicitor, we analyze an example system.

---

![Diagram of Client Layers of The Event Monitoring Architecture](image)

**Figure 7:** Client Layers of The Event Monitoring Architecture (Mellin 2004)

some libraries, are compiled into an executable file. Table 1 illustrates how the source file, header file and libraries can be combined when generating the executables according to figure 8.

![Diagram of Overview of Generation of Executables](image)

**Figure 8:** Overview of Generation of Executables

---

4.3 Expressing Constraints in Solicitor

To be able to use the event monitor as a test oracle, we must be able to specify correct and incorrect behaviours. To do this, the necessary constraints, such as deadlines, precedence relations or periods, must be considered.

To determine whether or not these factors can be expressed in Solicitor, we analyze an example system.

---

4The event monitor, test oracle and test object could either be compiled into separate executables or they could be compiled into a single executable.
4.3 Expressing Constraints in Solicitor

Table 1: Combinations of Files for Executable Generation

<table>
<thead>
<tr>
<th>Executable</th>
<th>Source Code</th>
<th>Event Monitoring Libraries</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Oracle (detection)</td>
<td>gen API</td>
<td>Source Test Oracle Source</td>
<td>Solicitor Event Monitor</td>
</tr>
<tr>
<td>Test Object</td>
<td>Test Object Source</td>
<td>Common gen API</td>
<td></td>
</tr>
</tbody>
</table>

We begin our work by creating a short example. From this example we analyze whether or not Solicitor’s operators are sufficient for expressing the necessary constraints.

Example 4: Analysis of an Example System

Table 2: Example Taskset

<table>
<thead>
<tr>
<th>Task</th>
<th>Priority</th>
<th>MIAT(^{a})</th>
<th>Offset</th>
<th>Deadline</th>
<th>WCET(^{b})</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>10</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>20</td>
<td>5</td>
<td>40</td>
<td>20</td>
<td>NULL</td>
</tr>
</tbody>
</table>

\(^{a}\)Minimum Inter Arrival Time  
\(^{b}\)Worst Count Execution Time

The taskset in table 2 gives us information about constraints in the system that is modelled. There are, for example, deadlines, a precedence relationship and priorities. To be able to monitor such a system we need to express these constraints using Solicitor.

As mentioned in section 2.4.3, Solicitor is an event specification language. We need a set of primitive event types to be able to express anything using Solicitor. Schütz (1994) gives a list of event types that are needed to monitor a real-time system. These event types include for example synchronization points, interrupts and access to time. In this report we emphasize synchronization points only. The reasons for this are given in
For this example, the following event types are deemed necessary:

- **SystemStart** — Generated when the system is started.
- **A start, B start** — Generated when tasks A & B respectively, start to execute.
- **A term, B term** — Generated when tasks A & B terminate.
- **A released, B released** — Generated when tasks A & B are released (that is, they are ready to start to execute).

Using these event types, we want to express the constraints, that can be derived from the taskset. Since meeting deadlines generally is an imperative constraint in the context of real-time systems, we start with trying to express the deadline constraint in Solicitor. There are several ways to do this:

- \( N(A_{\text{released}}, A_{\text{term}}, A_{\text{released}} + T) \)
- \( N(A_{\text{released}}, A_{\text{start}}, A_{\text{released}} + T) \)

### 4.3.1 Notation

In addition to the notation presented in section 2.4.3, we adopt the following notation, to make it easier to express the interesting constraints in Solicitor.

To be able to express time constraints, we define the term execution window. The execution window for a task \( i \) starts when task \( i \) is released, and stops when task \( i \) terminates (fig. 9). Hence, for the behaviour of the system to be correct, \( \forall i,j (w_{i,n}' \leq x_{i,n}' \land w_{i,n}'' \geq x_{i,n}'') \) must hold. That is, all tasks must start and terminate within their execution window.

- \( x_{i,n}' \) — Task \( i \), invocation \( n \) starts to execute.
- \( x_{i,n}'' \) — Task \( i \), invocation \( n \) terminates.
- \( w_{i,n}' \) — This expression states that the execution window of task \( i \), invocation \( n \) has started. This is equal to the release time of the task.
- \( w_{i,n}'' \) — This expression states that the execution window of task \( i \), invocation \( n \) has stopped. This is equal to the deadline of the task.
4.3 Expressing Constraints in Solicitor

Task
Released
Deadline
New
Release

Figure 9: Execution Window

- $x_{i,n}[r]$ — Task $i$, invocation $n$ locks the shared resource $r$.
- $x_{i,n}[r]$ — Task $i$, invocation $n$ unlocks the shared resource $r$.

This notation makes it easier to use different kinds of events in expressions written in Solicitor. For example, the following event expression uses this notation:

- $x_{i,n}', x_{i,n}''$

The first advantage of this notation is that it is comprehensive, we can express sufficient information in a brief expression. For example, we can see that the start of execution for task $i$ invocation $n$ is to be followed by the termination of task $i$ invocation $n$. The second advantage is that this notation has support for parameterized event types via the invocation index.

4.3.2 Precedence Constraints

Using Solicitor and the notations proposed in section 2.4.3 and section 4.3.1 we try to express precedence relation constraints. Four different cases\(^5\) has been identified:

- **Case 1**

  Figure 10 depicts the most basic case. The taskset consists of two tasks $i$ and $j$. The precedence constraint states that task $i$ must precede task $j$.

  A correct behaviour can be caught by the following expression:

  $- N(x_{i,n}', x_{j,n}, x_{j,n}, x_{i,n}'')$

\(^5\)Other cases can be transformed into combinations of these four cases.
4.3 Expressing Constraints in Solicitor

However, correct behaviours are often not interesting to monitor. Instead, we want to monitor and catch incorrect behaviours. The following expression would catch erroneous behaviours where the two tasks overlap:

\[-A(x'_{i,n}, x'_{j,n} \lor x''_{j,n}, x''_{i,n})\]

While the former expression catches all incorrect behaviours where the tasks overlap, it does not catch an incorrect behaviour where task \( j \) completely precedes task \( i \) (fig. 11).

To detect the erroneous behaviour depicted in fig. 11, we need a point in time, from which we can say, that it is an incorrect behaviour, if a task \( j \) starts to execute, if no task \( i \) has executed. To be able to do this, we need an interval that is initiated every time any task \( i \) or \( j \) starts to execute. The interval can be terminated \( T \) time after the interval was initiated (fig. 12).

Figure 10: Precedence Relation - Case 1

Figure 11: Precedence Relation - Case 1, Incorrect Behaviour

Figure 12: Precedence Relation - Monitoring Interval

Figure 12 depicts how such an interval could work. For this case (as described in fig. 10 and fig. 11), \( E = x'_{i,n} \lor x'_{j,n} \) and \( T \) is the deadline.
4.3 Expressing Constraints in Solicitor

associated with the initiating task. We can now catch the incorrect
behaviour when task \( j \) precedes task \( i \) with the following expression:

\[-x''_{j,n} \triangleleft x'_{i,n}\]

The expressions that are used to monitor these kinds of erroneous
behaviours, will hereby be referred to as \textit{general precedence expressions}.
It is necessary to monitor all the general precedence expressions, to be
able to catch all erroneous behaviours for all precedence constraints.

- **Case 2**

  In case 2 (fig. 13), case 1 has been extended with another task \( k \). In
  this case, task \( i \) precedes task \( j \) and task \( j \) precedes task \( k \). Transitivity
gives that task \( i \) precedes task \( k \).

  To catch all erroneous behaviors of case 2, we use the following
  expression, that catches all erroneous behaviours, when the tasks
  overlap:

  \[-A(x'_{i,n}, x'_{j,n} \triangleleft x''_{j,n} \triangleleft x'_{k,n} \triangleleft x''_{k,n}, x''_{i,n})\]

  together with the general precedence relation:

  \[-x''_{j,n} \triangleleft x'_{i,n} \triangleleft x''_{k,n} \triangleleft x'_{j,n}\]

- **Case 3**

  In case 3 (fig. 14), task \( i \) precedes both task \( j \) and task \( k \). There are
  no precedence relations between tasks \( j \) and \( k \).

  All incorrect behaviours are caught by the following expression:

  \[-A(x'_{i,n}, x'_{j,n} \triangleleft x''_{j,n} \triangleleft x'_{k,n} \triangleleft x''_{k,n}, x''_{i,n})\]

  together with the general precedence expression:
### 4.3 Expressing Constraints in Solicitor

- **Case 3**

In figure 14, tasks $i$, $j$, and $k$ precede task $i$. There are no precedence relations between tasks $j$ and $k$.

**Expression:**

\[-x''_{j,n} \vee x'_{i,n} \wedge x''_{k,n} \leq x'_{i,n}\]

- **Case 4**

In case 4 (fig. 15), tasks $j$ and $k$ precede task $i$. There are no precedence relations between tasks $j$ and $k$.

**Expression:**

\[-x''_{i,n} \vee x'_{j,n} \wedge x''_{i,n} \leq x'_{i,n}\]

**Expression:**

\[-A(x'_{j,n}, x'_{i,n} \vee x''_{i,n}, x''_{j,n}) \vee A(x'_{k,n}, x'_{i,n} \vee x''_{i,n}, x''_{k,n})\]

together with the general precedence expression:

\[-x''_{i,n} \vee x'_{j,n} \wedge x''_{i,n} \leq x'_{i,n}\]

#### 4.3.3 Time Constraints

When a dynamic real-time system is tested, there are several types of time constraints that are interesting to monitor. Three such types are *(i)* release times, *(ii)* execution times, and *(iii)* deadlines. All of these types are equal.
in the sense that we must be able to express a timeout to monitor them. In our work, we use the Solicitor relative temporal operator to accomplish this.

To monitor time constraints using a timeout, we can use the following general expression:

\[ N(\text{StartTime}, E, \text{StartTime} + T) \]

Where \( \text{StartTime} \) is the point in time from which the period of the deadline was initiated, \( E \) is the set of events, that are to occur within the deadline, and \( T \) is the length of the deadline interval.

The general timeout expression can then be tailored to catch, for example, missed deadlines:

\[ N(w_{i,n}', x_{i,n}', x_{i,n}'', w_{i,n}' + T) \]

### 4.3.4 Shared Resource Constraints

Real-time systems usually have several tasks that use shared resources. To be sure that these resources are used in a correct way by the tasks, we must monitor the constraints that are stated for the shared resources, in the formal specification. In our work, we assume that a resource must be exclusively locked (compare with binary semaphores), before it can be used by a task, and that it must be unlocked before it can be used again.

When we monitor constraints related to shared resources, we really monitor the events that are generated by the lock and unlock operations. For example, if a lock operation has occurred on resource \( r \), then no other lock operation can occur on that resource, until an unlock operation for that resource has occurred. If we have support for parameterized event types, the following expression catches all erroneous behaviours:

\[ A(x_{i,n,[r]}, x_{j,n,[r]} \triangledown x_{j,n,[r]}, x_{i,n,[r]}) \]

If we do not have support for parameterized event types, that is, we have no value for the invocation index \( n \), we also need the following expression\(^6\):

\[ A(x_{i,n,[r]}, x_{i,n,[r]}', x_{i,n,[r]}'') \triangledown A(x_{j,n,[r]}, x_{j,n,[r]}', x_{j,n,[r]}'') \]

\(^6\)We write out the invocation index \( n \) in the expression to be consistent, even though it will not be used.
5 Results

In this chapter, the results will be presented.

5.1 Need for Parameterized Event Types

In chapter 4, different constraints were presented together with composite event expressions that can be used to monitor these constraints. However, most of the expressions are not sufficient without support for parameterized event types.

Figure 16 illustrates how a recurring task may overlap with itself. This kind of overlap may occur whenever\textsuperscript{7} $MIAT < Deadline$.

Assume that the task in fig. 16 must complete before its deadline. The length of the execution window (deadline) is $T$ time. We use the following event expression to catch any incorrect behaviours:

$N(w'_{i,n}; x'_{i,n}; x''_{i,n}; w'_{i,n} + T)$

To simplify the event trace, we assume further, that all tasks will always finish within their execution window (and thereby within their deadline) so that no $w''_{i,n}$ events will occur. Then, we do not have to include the $w''_{i,n}$ events in the event trace. From fig. 16 we can now derive the following event trace:

$w_{i,n}; x_{i,n}; x'_{i,n}; x''_{i,n}$

In section 2.4.4, the recent context and the chronicle context was briefly described. Figure 17 depicts how the event trace would be composed in recent context.

\textsuperscript{7}If it is a single task, this can happen in a multi-processor environment, multiple tasks can overlap in a uni-processor environment if preemption is supported.
5.1 Need for Parameterized Event Types

In recent context, as Fig. 17 shows, the events would be composed in an incorrect way. This could lead to a correct behaviour being reported as an incorrect behaviour, or an incorrect behaviour not being detected. The reason for the incorrect composition is that the events are not parameterized on order.

If we have support for parameterized event types, and we use the chronicle context, the events would be composed as in Fig. 18. In this case, the events are composed in a correct way.

However, if we do not have support for parameterized event types, incorrect compositions may occur also in the chronicle context. Figure 19 depicts another example where a single task overlaps with itself. In this case, the following event trace is observed:

\[ w'_{i,n} ; x'_{i,n} ; w'_{i,n} ; x'_{i,n} ; x''_{i,n} ; x''_{i,n} \]

Figure 20 depicts how this event trace would be composed in the chronicle context. Again, we see that we need support for parameterized event types.

As illustrated in figure 16, even a single task may overlap with itself (of course multiple tasks may overlap with each other). Without support for parameterized event types, under certain conditions, events may be composed in incorrect ways and we might throw away needed events. Because of this, we need to have support for parameterized event types in the language of choice (in this case Solicitor). We could then parameterize the generated...
5.2 Precedence Constraints

In section 4.3.2 precedence constraints were discussed, and four different cases were presented, together with event expressions that can be used to catch erroneous behaviour. Without support for parameterized event types.

events on order, so that the events would be composed correctly and no needed events would be thrown away by the event monitor.

A more general case is illustrated in figure 21. Here we have three tasks that are to be executed in the same order. If two of the sequences overlap, we have the same problem as before, events may be composed incorrectly and we might throw away needed events. Hence we need parameterized event types.

5.2 Precedence Constraints

In section 4.3.2 precedence constraints were discussed, and four different cases were presented, together with event expressions that can be used to catch erroneous behaviour. Without support for parameterized event
5.3 Time Constraints

To express time constraints, if $MIAT < Deadline$, we also need support for parameterized event types. The reason for this is that even a single task may
overlap with itself (as shown earlier).

5.4 Shared Resource Constraints

As shown in section 4.3.4, shared resource constraints can be expressed without support for parameterized event types. However, with support for parameterized event types, one expression is enough, otherwise, two expressions are needed.
6 Discussion

In this chapter, the results presented in chapter 5 are discussed.

6.1 Formal Specification is Insufficient

When tasks or sequences of tasks, can overlap (as described in section 5.1), we can not really know what a precedence relation means.

![Figure 22: Precedence Constraints - Overlapping Tasks](image)

Figure 22 depicts a system that consists of two recurring tasks $i$ and $j$. A precedence constraint states that task $i$ precedes task $j$. From this constraint, we can not know if the case depicted in figure 22, is a correct or incorrect behaviour. Depending on the target system, it could be either correct or incorrect. The formal specification must be extended to describe what is a correct and incorrect behaviour in such a case.

6.2 Ordering of Event Occurrences

If the event specification language Solicitor and the event monitor architecture are to be extended with parameterized event types, we must consider, how to order the event occurrences (if we want to parameterize them on their order).

Four different strategies for ordering the events are described here. First, ordering can be global or local, with respect to event types. Global ordering is valid for all events on the monitored system, while local ordering is valid for the current event expression. Secondly, ordering can be absolute or relative. Relative ordering only takes events that have not been thrown out of the filtered event log (Mellin 2004, pp. 12 – 13). Absolute ordering, on the other hand, takes all event occurrences into consideration.
These can be combined into four different ordering strategies (Jonas Mellin, personal communication, 2005):

- Absolute Global Order
- Absolute Local Order
- Relative Global Order
- Relative Local Order

To illustrate the differences between absolute and relative order, a small example will be used:

Example 5: **Absolute & Relative Order**

Assume that we have two composite event expressions, $E_A$ and $E_B$, such that: $E_A = E_{i,n}; E_{j,n}$ and $E_B = E_{j,n}; E_{k,n}$

Assume that the following event trace is generated: $E_{j,1}; E_{i,1}; E_{j,2}$

If the first event is consumed, the ordering in absolute order would be: $E_{j,1}; E_{i,1}; E_{j,2}$, and in relative order it would be: $E_{i,1}; E_{j,1}$

6.3 **Notation Support for Parameterized Event Types**

In chapters 4 and 5 we have shown that, in some cases, support for parameterized event types is necessary. Event though Solicitor and the event monitoring architecture do not currently support parameterized event types, the notation proposed in section 4.3.1 does have support for this.
7 Related work

This chapter describes related work.

Savor & Seviora (1998) utilize a software supervisor to perform automatic detection of software failures. Their supervisor has similarities Peters & Parnas’s monitor (Peters & Parnas 2002), and Mellin’s event monitor (Mellin 2004).

Both Mellin, Peters & Parnas, and Savor & Seviora use a model of correct behavior, derived from the target system’s requirements specification (formal specification in our work). All approaches monitor the inputs and outputs of the target system. Peters & Parnas and Savor & Seviora reports deviations between observed and specified behaviour as failures.

Both Peters & Parnas’s monitor and Savor & Seviora’s supervisor can be used as test oracles, while we in our work have investigated how Mellin’s event monitor can be extended with (the functionality of a) test oracle.

Nor Peters & Parnas’s approach or Savor & Seviora’s approach instrument the test object (the target system). Mellin however, does this.

Savor & Seviora use FIFO queues to store expected and observed behavior, Mellin uses a filtered event log to store event occurrences.

Jahanian et al. (1994), have support for parameterized event types in their notation. Our work have investigated how to extend Mellin’s event monitor and event specification language Solicitor with this.
8 Conclusions

This chapter briefly summarizes our work, the main contributions that this work has made are listed, and interesting future work is proposed.

8.1 Summary

This project emphasizes how event monitoring can be used in test oracles to conduct automated testing.

An event monitor can be used as a detector in an automated test oracle to monitor the target system during testing. The event monitor observes the behaviour of the target system by using sensors that register event occurrences. If the event monitor detects any deviations between the behaviour observed on the target system and the model of correct behaviour derived from the formal system specification, it signals this to a reporter. The reporter then takes some action, for example it may propagate the signal to the person that is conducting the test.

In order to use the event monitor as a detector in an automated test oracle, we must be able to express the constraints that the event monitor should monitor. The event monitor can then use these expressions to distinguish between positive and negative results. We express expected behavior as composite event expressions in the event specification language Solicitor. The constraints that the event expressions are used to model, are derived from the formal specification.

This kind of testing has advantages over conventional testing (manual testing). When testing a dynamic real-time system, the amount of test cases that have to be executed and evaluated makes it impractical to do manually. The reason for this is that we must take the time at which event occurs into account when we test dynamic real-time systems. Further, regression testing is hard to do manually compared with doing it automatically. The reason for this is that a dynamic real-time system is dynamically scheduled and event-triggered, and therefore, it is hard to reproduce test case executions.

Another benefit with using event monitoring and automated test oracles is that we do not need to know much about the target systems behaviour in advance, we just monitor its behaviour and analyze whether the executed test case gave a positive or negative result. These factors together implies that automated testing using event monitoring and automated test oracles is superior compared to conventional testing.

We have shown that in order to be able to express all the constraints in the formal specification, we must have support for parameterized event types. With support for parameterized event types we can express all the constraints
that are analyzed in this report. Without support for parameterized event types we can express special cases (e.g., when $MIAT \geq Deadline$) but this is probably not enough in most cases. For these reasons we conclude that support for parameterized event types is a significant requirement for an event specification language that is to be used in event monitoring. However, most event specification languages lack support for parameterized event types, Solicitor is no exception. It would therefore be interesting to extend Solicitor with support for parameterized event types in the future. Also, when new event specification languages are developed, it is important that support for parameterized event types is at least considered. The reasons for this have been described above.

8.2 Contributions

The main contributions of this work are as follows:

- We have shown that support for parameterized event types is a minimal requirement in Solicitor (and probably in any event specification language that is to be used for event monitoring).
- We have provided an overall architecture for test oracle management.
- We have listed some common constraints, and shown how event expressions can be constructed to detect erroneous behaviour.
- We have suggested a comprehensive event occurrence notation.

8.3 Future Work

- **Extending Solicitor with Support for Parameterized Event Types**
  In this work, we have demonstrated that parameterized event types are necessary to be able to express all constraints that are modelled in the formal specification. Hence, it is necessary to extend the event specification language Solicitor with support for this.

- **How to Generate Source Code for Test Oracle Reporter**
  Since this work emphasize detection functionality of a test oracle, it would be interesting to investigate how the report functionality should be generated.

- **Early Detection of Erroneous States**
  According to Jahanian et al. (1994): “Detecting violations as early as
possible is a desirable property, because it can allow the system to take corrective action before the violation actually happens.”

Therefore, it would be interesting to investigate how event expression should be constructed so that they always detect erroneous behaviours as soon as possible. An extension to this would be to detect behaviour that could lead to erroneous behaviour.

- **Optimizing Event Expressions**
  It would be interesting to investigate if there exists an optimized or minimal event expression, that would catch all erroneous behaviours for a specific constraint. If such an expression always existed, it could be used to minimize the set of expressions needed to monitor the system. This could lead to more effective monitoring. Further, if we knew that the optimized expression was complete (i.e., it covers all erroneous behaviours) we would know that the event monitor and test oracle, would detect all erroneous behaviours.

- **Algorithmic Approach to Optimize Expressions**
  If there exists an optimized or minimal event expression it would be interesting to investigate if it exist an algorithm that can be used find that expression for a given constraint. It would also be interesting to show how such an algorithm could be implemented.
References


A  Symbols & Style Conventions

The symbol ■ will be used to indicate that the above example has ended, accordingly, the symbol □ will be used to indicate that the above definition has ended.

B  Figure Legends

- Process
- Component, Object
- Single file
- Multiple files