Verifying transformations between timed automata specifications and ECA rules

HS-IDA-MD-03-002

AnnMarie Ericsson

Submitted by AnnMarie Ericsson 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 2003

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

__________________________
AnnMarie Ericsson
Abstract

Event-triggered real-time systems are desirable to use in environments where the arrival of events are hard to predict. The semantics of an event-triggered system is well mapped to the behaviour of an active database management system (ADBMS), specified using event-condition-action (ECA) rules. The benefits of using an active database, such as persistent data storage, concurrency control, timely response to event occurrences etc. highlights the need for a development method for event-triggered real-time systems using active databases.

However, there are problems left to be solved before an ADBMS can be used with confidence in real-time environments. The behaviour of a real-time system must be predictable, which implies a thorough analysed specification with e.g. specified worst case execution times. The predictability requirement is an obstacle for specifying real-time systems as ECA rules, since the rules may affect each other in many intricate ways which makes them hard to analyse. The interaction between the rules implies that it is not enough to verify the correctness of single rules; an analysis must consider the behaviour of the entire rule set.

In this dissertation, an approach for developing active applications is presented. A method is examined which starts with an analysed high-level timed automaton specification and transforms the specified behaviour into an implicitly analysed rule set. For this method to be useful, the transformation from timed automata to rules must preserve the exact behaviour of the high level specification. Hence, the aim of this dissertation is to verify transformations between timed automaton specifications and ECA rules.

The contribution of this project is a structured set of general transformations between timed automata specifications and ECA rules. The transformations include both transformations of small timed automata constructs for deterministic environments and formally verified timed automata patterns specifying the behaviour of composite events in recent and chronicle context. 

Keywords: Active rules, Timed automata, Real-time systems, Transformations
# Contents

1 Introduction ................................. 1
   1.1 ECA rules versus timed automata ......................... 1
   1.2 Project approach ....................................... 3
   1.3 Results ............................................. 4
   1.4 Outline of this project ................................. 4

2 Background .................................. 6
   2.1 Real-time systems ........................................ 6
      2.1.1 Event-triggered versus time-triggered systems ........... 7
      2.1.2 Real-time databases .................................. 8
   2.2 Active databases ........................................ 9
      2.2.1 ECA rules ........................................... 10
      2.2.2 Composite events ..................................... 11
      2.2.3 Event consumption policies ............................. 12
      2.2.4 Execution model ..................................... 15
   2.3 Active applications ................................. 17
      2.3.1 Termination ......................................... 17
      2.3.2 Confluence ......................................... 19
      2.3.3 A Distributed Active Real-time Database System (DeeDS) ... 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4</td>
<td>Formal methods</td>
<td>21</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Proof of correctness</td>
<td>21</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Benefits of formal methods</td>
<td>22</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Formal schema for composite events</td>
<td>22</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Finite-state machines and Finite automata</td>
<td>26</td>
</tr>
<tr>
<td>2.4.5</td>
<td>Timed automaton</td>
<td>29</td>
</tr>
<tr>
<td>2.4.6</td>
<td>Uppaal</td>
<td>31</td>
</tr>
<tr>
<td>2.5</td>
<td>Timed automata to ECA rules</td>
<td>34</td>
</tr>
<tr>
<td>3</td>
<td>Problem description</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>Active real-time database applications</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Deriving active rules from timed automata</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Aim and objectives</td>
<td>40</td>
</tr>
<tr>
<td>3.4</td>
<td>Limitation of project scope</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>Method</td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>Generalize transformations and identify limitations of transformations</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Specify composite events in timed automata</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Formally verify equivalence of semantics</td>
<td>46</td>
</tr>
<tr>
<td>5</td>
<td>Verify transformations</td>
<td>48</td>
</tr>
<tr>
<td>5.1</td>
<td>Generalize transformations and identify limitations of transformations</td>
<td>48</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Enter state</td>
<td>49</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Transition assignments</td>
<td>50</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Guards</td>
<td>51</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Time constraints</td>
<td>53</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Synchronisation</td>
<td>56</td>
</tr>
</tbody>
</table>
5.1.6 Limited transformation of guards ............................. 58
5.2 Specifying patterns in timed automata ........................... 66
  5.2.1 Conjunction ............................................. 66
  5.2.2 Disjunction .............................................. 73
  5.2.3 Sequence ................................................ 75
5.3 Equivalence of semantics ........................................... 79
  5.3.1 Assumptions ................................................ 80
  5.3.2 Chronicle Context Predicate .................................. 81
  5.3.3 Operator predicate ......................................... 84
  5.3.4 Verifying conjunction pattern ................................. 85
  5.3.5 Verifying disjunction pattern .................................. 95
  5.3.6 Verifying sequence pattern .................................... 99

6 Analysis ..................................................................... 107
  6.1 General transformations ........................................... 107
    6.1.1 Transforming constructs of timed automata from a deterministic environment ........................................... 107
    6.1.2 Limitations of transformations ................................ 109
    6.1.3 Prerequisite for transformations ................................ 111
  6.2 Composite events specified in timed automata ................. 111
    6.2.1 Applicability of timed automata patterns ..................... 112
    6.2.2 Specification and implementation issues ....................... 113
  6.3 Formally verify equivalence of semantics ......................... 114
    6.3.1 Choice of formal notation ................................... 114
    6.3.2 Equivalence of behaviour ..................................... 115
  6.4 Related Work ..................................................... 115
7 Conclusion

7.1 Project summary .................................................. 121
7.2 Discussion ......................................................... 122
  7.2.1 High level specification .................................. 122
  7.2.2 Transformation issues .................................... 123
  7.2.3 Resulting rule set ......................................... 124
7.3 Project conclusion ................................................ 127
7.4 Contributions .................................................... 128
7.5 Future work ...................................................... 129
Chapter 1

Introduction

Event triggered real-time systems, which are exposed to sporadic and a periodic event occurrences, may take advantage of the ability to handle reactive behaviour in an active database management system (ADBMS). In traditional database systems, the semantics of monitoring changes in the database is distributed, replicated and hidden in different applications using the database. In contrast, the reactive behaviour of an ADBMS is moved from the applications into the ADBMS, making it possible to monitor and react to specific event occurrences in a centralized and timely manner (Paton & Diaz 1998).

1.1 ECA rules versus timed automata

The behaviour of an active database is often specified as a set of low-level executable ECA (Event, Condition, Action) rules, where an event is triggering an action if a certain condition is true. However, the ability to specify a large real-time system, with high predictability requirements, using the ECA rule paradigm, is limited due to the difficulty of analyzing the behaviour of a large rule set.

The low analysability of a set of rules is mainly caused by the rules ability to interact
CHAPTER 1. INTRODUCTION

with each other in many intricate ways. An executing rule may for example trigger another rule, which updates some data objects that in turn causes the original rule to be triggered. Behaviours like this can form an infinite loop of cascade triggering that causes a non-terminating behaviour. Another problem is to determine if the set of rules is confluent, i.e. if the outcome of simultaneously fired rules depends on in which order the rules were executed. If the rule set is not confluent, there may be different kinds of race conditions, where the outcome of the condition evaluation (or action execution) of two simultaneously triggered rules is dependent on in which order their conditions were evaluated or in which order their actions were executed (Paton & Diaz 1998).

The difficulty of analyzing the behaviour of a set of ECA rules is an obstacle for using active databases to develop active real-time applications. Real-time systems are required to respond to external stimuli within a finite and known time-period. The correctness of a real-time system is not only depending on its logical correctness, but also on the ability to meet its specified time constraints. The requirement on predictable response times is hard to fulfill as the rule base is on a too low level of abstraction to be thoroughly analyzed. There is also a very limited access to CASE tools supporting the area of ECA rule development.

Since real-time systems are frequently used for monitoring and controlling objects in environments where failures may lead to a disaster, the use of formal methods is desirable in the specification phase of such systems. Formal methods are based on mathematics and the benefits of using them is that the specification produced are unambiguous. It is also possible to prove that certain characteristics are met in the system and that the implementation of the system meets its specification. An example of a formal method which is designed to handle the modelling of real-time systems is timed automata, which is a finite automaton extended with a set of clocks (Alur & Dill 1994). If a system is specified using timed automata, it is possible to verify characteristics like absence of deadlock and that a certain system state will be reached within a specified time period. A CASE tool with
CHAPTER 1. INTRODUCTION

model checking capabilities may be used for automatic verification of such characteristics. One CASE tool developed for this purpose is Uppaal (Larsen, Pettersson & Yi 1997) which also provides the possibility of graphical simulations of timed automata models.

1.2 Project approach

The desire to use active databases in real-time systems is one of the issues which highlight the need for a systematic method for developing an analysed set of active rules with predictable behaviour. Our assumption is that transforming executable ECA rules from a formally verified specification, such as timed automata, preserving the specified behaviour during the transformation to rules, results in an implicitly analysed set of rules. This requires that the exact behaviour of the formal model is transformed; no additional behaviour that affects the operation of the system is allowed to be introduced during the transformation. It is also desirable that it is possible to reverse engineer the formal model from the rule set.

The high level aim of this project is to take advantage of an analysed formal specification and transform its behaviour into active rules. This is not a new idea; it is the same idea as transforming a C++ implementation to assembler, and the approach has been used in previous projects concerning ECA rules, for example Berndtsson, Chakravarthy & Lings (1997) who explores the ability to derive ECA rules from finite automata, and Falkenroth & Törne (1999) who has constructed a compiler for this purpose. The uniqueness of this project compared to the previous approaches is the focus on real-time systems, as well as the use of a high level specification language with the ability to express timeliness requirements.

Since constructing an entire rule compiler does not fit into the time frames of this project, the aim to be reached in this project is to verify transformations between timed
automata and ECA rules. Besides adding time constrains to the transformation process, the focus of real-time systems are used as an argument for limiting the scope of the target execution model in this project.

1.3 Results

The result of this project is a set of transformations from different timed automata constructs to rules. Constructs of timed automata that are hard to transform into rules are also identified and alternative solutions are suggested. To facilitate specification and transformation of composite event occurrences, the behaviour of composite events in recent and chronicle context is specified as timed automata patterns. The behaviour of composite events is also expressed as regular expressions where it is possible, which will facilitate the identification of a composite event in an arbitrary timed automaton specification. The timed automata patterns of composite events for chronicle context are formally verified to have an equal behaviour as the composite events in the rule set for any sequence of inputs.

1.4 Outline of this project

Chapter 2 gives a background about theories behind timed automata and active rules, as well as a brief introduction to real-time systems, active databases, formal methods, timed automata, Uppaal and DeeDS. A presentation of a method for transforming rules between timed automata and ECA rules developed by Ericsson (2002) is presented as well, since the work in this project is based upon this transformation method.

In chapter 3 a problem description is given as well as a presentation of the aim specified for this project and objectives identified to reach this aim. In chapter 4 the method used to reach the objectives are described and in chapter 5 and 6 project results and an analysis
of the results are presented. Chapter 7 presents a summary of results, discussion around the results, related work, contributions and future work.
Chapter 2

Background

The following sections will give the background knowledge required to understand the problem to be solved and concepts used in the result part of this dissertation.

2.1 Real-time systems

Besides being logically correct, a real-time system must also meet its timeliness requirements. The system must be able to answer to external stimuli within a specified time period, as well as meeting requirements on possible delay times.

Depending on the consequences of missing a deadline, real-time systems may be classified into hard (hard essential and hard critical), firm and soft real-time systems. In a hard real-time system the consequences of missing a deadline is catastrophic, leading to considerable damage e.g. loss of human lives (hard critical) or considerable economical loss (hard essential). In firm real-time systems the consequences of missing a deadline are the loss of service and in a soft real-time system, a task that has missed its deadline may still produce some value to the system (Eriksson 1997).

An important characteristic of a real-time system is timeliness. Timeliness means that
a system has predictable response times and is sufficiently efficient. This implies that each task in the system must have predictable and sufficient resource requirements (e.g. memory, bandwidth etc.).

2.1.1 Event-triggered versus time-triggered systems

Depending on how the system interacts with its environment, there are two main design approaches to use in the area of real-time systems (Kopetz & Verissimo 1993). In hard real-time systems, where the environment is monitored in time periods, the time triggered approach is most common. In every period the system is checking the state of the environment and depending on the current state of the environment, it takes the appropriate action. The advantage of the time triggered approach is that it can be thoroughly analysed a priori. The worst case execution times of the system can be specified, resulting in a system with predictable response times on each task. The main disadvantages are that the system is always allocating resources according to its specified worst case behaviour, even in cases where the average resource usage is much less and that sporadic tasks are hard to handle. It is also the case that all systems do not lend themselves to the thorough a priori analyses that is required in time-triggered systems. There may be too little knowledge about the systems behaviour a priori, for example if the system is monitoring an unpredictable environment.

The other major design approach for real-time systems is the event triggered approach, where the system is required to handle events occurring anytime. Such systems are idle (or e.g. performing some background task) waiting for an event to occur and as an event occurs, the system is immediately responding to it. Event triggered systems are harder to analyse than time triggered systems, since there are an almost infinite number of possible different execution traces in event triggered systems. This makes it hard to calculate the worst case response times for each task and the desired predictable behaviour is harder to
CHAPTER 2. BACKGROUND

guarantee. The advantages of event triggered systems are that they can handle a variety of
tasks whose execution order is not known a priori. In contrary to time triggered systems,
they can handle overload situations without falling apart. In this dissertation, the event
triggered approach is assumed as real-time systems are concerned.

2.1.2 Real-time databases

There are several features supplied by a database management system that is advantageous
to use in real-time applications. Specifically, a database schema helps to avoid redundancy
of data and its description, transaction support ensures correctness of concurrent transac-
tion executions and ensures data integrity maintenance, even in the presence of failures,
etc. (Ramamritham 1993).

However, real-time systems often have high predictability requirements. Its worst case
execution times and resource usage must be known. Unfortunately, the use of databases
adds a number of sources of unpredictability. According to Ramamritham (1993) the
execution traces of transactions can depend on the data values, transactions can abort
resulting in rollbacks and restarts, there may be data and resource conflicts and there may
be unpredictable I/O requests.

A second problem is to keep the data in the database consistent with the controlled
environment, since real-time systems are frequently used to control environment external
to the system, for example robots in a factory. In such scenarios it is important that
the internal state of the system is consistent with the corresponding external state of
the controlled environment. Otherwise the consequences may be disastrous. Imagine
for example the consequences if a system, controlling a robot which moves towards an
expensive target in high speed. The robots actual distance to the target is 5 centimetres
but according to the system state, the distance is 10 centimetres. The robot will probably
crash into the target instead of being able to perform its task, destroying both the target
and the robot. This exemplifies the importance of keeping the system consistent with the environment, and that some data, for example distances, are only valid for a specific period of time. This implies that such systems have timing constraints arising from the need to continuously track the environment, however, timing constraints also arises because of the need to make data available to the controlling system for its decision making activities (Ramamritham 1993).

2.2 Active databases

Traditional database management systems (DBMS) are passive, meaning that they do not automatically react to changes in the database. In such systems a request (for example update or query) is only executed if it is explicitly raised by an application using the database. An active database on the other hand is automatically reacting to specific changes in the system and performs some predefined action as these changes occur.

If it is desirable to use active behaviour in a passive database system, then there are two possible approaches to achieve this. Either the active semantics is implemented in each application using the database, or a polling mechanism is used to periodically check the database for changes. However, if the active behaviour is implemented in each application, the monitoring functionality is distributed, replicated and hidden among different applications. This is likely to be a problem when it comes to system maintenance. Using the polling mechanism makes it possible to represent the semantics in one single place. However, the frequency with which the database is polled is a problem here. Polling the system too often causes unnecessary load, while polling too seldom causes the risk of missing that something important has occurred (Paton & Diaz 1998).
2.2.1 ECA rules

In an active database system the reactive behaviour is moved from the application (or polling mechanism) into the database management system. In this way the reactive behaviour is centralized and handled in a timely manner (Paton & Díaz 1998).

The behaviour of a system may be specified using active rules, described as ECA rules. ECA rules consists of up to three components; events, conditions and actions. The event part specifies the event occurrence on which the rule is triggered. The condition specifies a condition which must be true for the action to be executed and the action part specifies which action to be performed as the event has occurred and the condition is evaluated to true. If the event part is left out, the resulting rule is a production rule (CA) and if the condition part is missing, the resulting rule is an event-action (EA) rule.

The event part of the ECA rule may be primitive or composite. A primitive event occurrence is something that happens at a point in time and is raised by a single occurrence, for example an update in the database, a specified clock time, or an external event occurrence raised by a happening outside the database. A composite event is raised by a combination of primitive or composite events. The occurrence of an event can be described as a predicate. The $O(E, [t, t'])$ predicate introduced by Galton & Augusto (2001) is true if an event $E$ has occurred that start at time $t$ and terminate at time $t'$. In this dissertation, the notation $occ(E, [t, t'])$ introduced by Mellin (2003) will be used instead of $O(E, [t, t'])$ since $O$ can be confused with the big-oh notation for algorithmic complexity. Formally, the interval function $[t, t']$ states that $start([t, t']) = t$, $end([t, t']) = t'$ and $|[t, t']| = t' - t$ (Mellin & Andler 2002).
2.2.2 Composite events

A composite event type may be combined by different operators, like for example conjunction, disjunction or sequence. The following models describes the occurrence of composite events formally (Galton & Augusto 2001). $\mathcal{G}$ is a model of the history of events that have occurred.

**Primitive event occurrence**

Let $\mathcal{G}$ be a model such that:

$$\mathcal{G} \models occ(E, [t, t]) \text{ if } \text{ prim}(E) \text{ and such an event type has occurred at time } t.$$ 

In other words, a primitive event of type $E$ has occurred.

**Disjunction ($\lor$)**

The disjunction operator can for example be used to specify that the composite event $E$ occurs if an event of type $E_1$, or $E_2$ occurs within a specified time period.

$$\mathcal{G} \models occ(E_1 \lor E_2, [t, t']) \text{ if } \text{ occ}(E_1, [t, t']) \text{ or } \text{ occ}(E_2, [t, t'])$$

**Sequence ($;$)**

The sequence operator can be used to specify that a composite event of type $E$ occurs if a set of other events occur in a specified sequence. The following model expresses that the composite event of type $E$ is raised if an event of type $E_1$ occurs before an event of type $E_2$ and that both event occurrences occurs within the time period that starts with $t$ and ends with $t'$. 

11
\[ \mathcal{G} \models \text{occ}(E_1; E_2, [t, t']) \iff \exists t_1 \leq t_2 (\text{occ}(E_1, [t, t_1]) \land \text{occ}(E_2, [t_2, t']) \land (t_1 - t > 0 \lor t' - t_2 > 0)) \]

**Conjunction (\(\triangle\))**

The conjunction operator can for example be used to specify that a composite event of type E is raised if there is an occurrence of event \(E_1\) and \(E_2\) within a specified time period.

\[ \mathcal{G} \models \text{occ}(E_1 \triangle E_2, [t, t']) \iff \exists [t_1, t_2] ((\text{occ}(E_1, [t, t_1]) \land \text{occ}(E_2, [t_2, t'])) \lor (\text{occ}(E_2, [t, t_1]) \land \text{occ}(E_1, [t_2, t'])) \\
\lor (\text{occ}(E_1, [t, t']) \land \text{occ}(E_2, [t_1, t_2]) \land t \leq t_1 \lor t_2 \leq t') \\
\lor (\text{occ}(E_2, [t, t']) \land \text{occ}(E_1, [t_1, t_2]) \land t \leq t_1 \land t_2 \leq t')) \]

**Non-occurrence (N):**

The non-occurrence event occurs if there is no event occurrence of a specified event, between the occurrences of two other specific events. In the example below, there is a non-occurrence of an event of type \(E_2\) if the occurrence of type \(E_2\) does not occur in the interval opened by the occurrence of an event of type \(E_1\) and closed by the occurrence of an event of type \(E_3\).

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

### 2.2.3 Event consumption policies

As a composite event is detected, there may be several different event occurrences of the same event type that can be used to form the composite event. An event may carry parameters, for example from the activity causing the event, or selected parts of the database state.
CHAPTER 2. BACKGROUND

at the event occurrence. Since parameters carried by event occurrences of the same type may be different, causing different results, it is important to consume event occurrences according to a predefined policy. Some frequently used consumption policies are recent, chronicle, continuous and cumulative. The different consumption policies can also be denoted as recent, chronicle, continuous and cumulative parameter contexts, as described in Chakravarthy & Mishra (1994), which will be used instead of consumption policy in this report.

Each composite event has a terminator and an initiator event. The initiator event initiates the detection of the composite event occurrence and the terminating event terminates the event composition. If for example the composite event is a sequence of type $E = E_1; E_2; E_3$ then $E_1$ is the initiator and $E_3$ is the terminator type for this specific composite event. Depending on which context that is concerned, the initiator and terminator have different meanings. An occurrence of a terminator event in the continuous context may for example raise several instances of a composite event type, while only one instance is raised by the same event sequence in the chronicle context.

In the continuous context, a composite event is initiated each time an initiating event (event that starts the detection of the composite event) occurs and the cumulative context accumulates all the primitive events until the composite event is raised (Paton & Diaz 1998).

This project is focusing on real-time systems and will only cover recent and chronicle context. Recent context is useful as for example the pressure and temperature of a liquid in a tank is measured. To discover hazardous situations, the pressure must not increase a certain value as the temperature is above a threshold value. For the measurements to be useful, only the most recent values are interesting. Chronicle context on the other hand is useful as the order of occurrence is important. It may for example be useful if two sensors placed on a railway track is measuring the time when train passes by. The earliest unused
time stamp of the first sensor should be matched with the earliest unused time stamp of
the second sensor to calculate the average speed of a certain train.

Recent context

In recent context, the most recent set of event occurrences are considered. Each terminator
is raising a composite event occurrence, even if some of the event occurrences it contains
have taken part in another instance of a composite event. As an illustrating example, let
$E_1$, $E_2$ and $E_3$ be primitive event types. $E_4$ is a composite event type which instance is
raised by the occurrence of events of type $E_2$ and $E_3$ ($E_4 = E_2 \triangle E_3$), $E_5$ is a composite event
type raised by the occurrence of an event of type $E_1$ and $E_2$ in a sequence ($E_5 = E_1; E_2$).

In an example scenario, events of type $E_1$ occurs at time $t_1$ and $t_2$, at time $t_3$ an event
of type $E_2$ occurs followed by an event of type $E_3$ at time $t_4$ and an event of type $E_2$ at
time $t_5$ where $t_1 < t_2 < t_3 < t_4 < t_5$. The composite events $E_4$ and $E_5$ is raised as in
Figure 2.1.

The first instance of type $E_4$ is raised as an event of type $E_3$ occurs for the first time,
because then there is an occurrence both of type $E_2$ and $E_3$ in the event history. As an
event of type $E_2$ occurs for the second time, there is a new instance raised of event type
$E_4$, since there already is an event of type $E_3$ in the event history. Both type $E_2$ and $E_3$
are terminators in this composite event, and each time a new event of type $E_2$ (or $E_3$)
occurs, an event of type $E_4$ will also be raised if there is an prior instance of $E_3$ (or $E_2$) in
the event history. As a new primitive event occurs in recent context, the old instance of
this event is overwritten and only the most recent occurrence is saved.

For the composite event of type $E_5$ to be raised, the primitive event occurrence of type
$E_2$ must be raised after an event of type $E_1$. This means that only $E_2$ is terminator type
in the composite event of type $E_5$, and a new instance of type $E_5$ will be raised whenever
there is an occurrence of type $E_2$ after an event occurrence of type $E_1$ in the event history.
CHAPTER 2. BACKGROUND

Figure 2.1: Occurrence of event $E_4 = E_2 \triangle E_3$ and $E_5 = E_1; E_2$ in recent context

In Figure 2.1 the event of type $E_5$ is raised both times as the event of type $E_2$ occurs.

Chronicle context

In chronicle context, event occurrences are consumed in chronicle order and each event occurrence does only participate in one composite event instance of each type. An event occurrence is invalidated if it is not of interest for any composite event. If events occur in an identical sequence as in the previous example, the composite events of type $E_4$ and $E_5$ are raised according to Figure 2.2. As opposed to the recent context, the second occurrence of type $E_2$, does not raise an occurrence of type $E_4$ since the earlier occurrence of type $E_3$ is consumed by a previous instance of $E_2 \triangle E_3$. The occurrences of type $E_5$ will use the earliest unused occurrence of $E_1$ instead of the most recent as in the previous example.

2.2.4 Execution model

The types of events, operators and contexts available belong to the knowledge model of the ADBMS. The knowledge model describes what can be said about the rules in an active database. On the other hand, the way a set of rules is treated at run time is determined
by the execution model of the active database. The execution model is among other things
describing which coupling modes are used, transition granularity, net effect policy and
cycle policy. The cycle policy is determining what happens when an event occurrence is
signalled by the evaluation of a condition, the executed rule may either be interrupted
by the signalled rule, or continue to execute causing the newly triggered rule to wait.
The transition granularity is determining whether a rule is triggered by a set or a tuple
of event occurrences and the net effect policy determines whether an event occurrence
is concerning single occurrences (e.g. update) or if the net effect of several occurrences
should be considered (e.g. only delete if update is followed by delete on the same data
item) (Paton & Diaz 1998).

The coupling modes determine when the condition is evaluated with respect to the event
occurrence and when the action is executed with respect to the evaluation of the condi-
tion. The most frequently supported options for coupling modes are; immediate where
the condition is evaluated immediately after the event occurrence (action executed imme-
diately after the condition evaluation), deferred where the condition (action) is evaluated
(executed) in the same transaction as the event occurrence although not necessarily immediately, and detached where the condition (action) is evaluated (executed) in a different transaction (Paton & Diaz 1998).

The semantic of an active rule is depending on the execution model of the ADBMS in which it is processed. This means that the same rule can behave differently in different ADBMS, which makes it important to take the target execution model into account when analysing the behaviour of a set of active rules.

2.3 Active applications

One of the main problems with the use of active rules is the difficulty of analysing the behaviour of an application implemented as a set of rules. The rules may depend on each other in ways that are hard to predict and hard to analyse. It may for example be important to restrict the order in which conditions of simultaneously triggered rules are evaluated, since the condition evaluation of one rule may affect the outcome of the condition evaluation of another rule.

2.3.1 Termination

A rule cascade occurs as the processing of one rule action is triggering another rule. If the cascade triggering is forming a circle as viewed in Figure 2.3, there are no guarantees that the set of rules will terminate. (In Figure 2.3 the processing of rule R1 is triggering the rule R2, which in turn triggers rule R3, which triggers rule R1 and a rule cascade circle is formed.)

The cascade of rule triggerings may not be only due to the action of one rule triggering another rule. Consider the case where the action of rule R1 causes the condition of a second rule R2 to become true, and a third rule R3 triggers the rule R2. In this case, the
CHAPTER 2. BACKGROUND

Figure 2.3: Example of nonterminating cascade triggering

first rule (R1) which satisfied the condition of rule R2 is partly responsible for the cascade triggering. This implies that it is not sufficient to study isolated effects of rule actions, since in isolation, these rules would not have caused a cascade, but their combined effect may cause rule cascades. An accurate capture of cascades must involve an investigation of complete rule sets in combination with possible sequence of database states, external updates, and rule processing semantics (Falkenroth 2000, p.151).

According to (Vaduva 1999, p.69) there are three possible ways to solve the termination problem. Firstly there is a run time solution which is handling the termination problem during the execution of the rules. In this solution, the system is assuming that infinite cascade triggering is taking place, and terminates the execution of rules if a certain number of rules are triggered. The drawbacks of this solution is that a set of rule triggerings might be terminated although it was performing perfectly correct, on the other hand, an incorrect circular triggering may unnecessarily run until the bound of rule triggering is reached. The second solution for the termination problem is to impose significant syntactic limitations to rule specification. However, this is reducing the expressive power of the rule specification language. Thirdly, the rule set may be analysed to detect subsets of rules which behaviour may lead to non-termination. These rules must be modified, and the termination analysis must be restarted. This is an iterative process performed until a terminating rule set can be guaranteed.
However, a complete prediction of the interactions in a set of ECA rules is identified by both (Vaduva 1999, p.83), and Falkenroth (2000) to be an undecidable problem. It is impossible to assert with certainty whether an interaction between rules will take place considering all database states and for all possible action statements. This is why the approach for solving this problem has to contain some restriction, or simplification of the problem, for example to exclude the state of the database in the analysis, or to raise the abstraction of the analysis, as performed in (Vaduva 1999).

An important note is that non-termination is not always an undesired property. As already pointed out by (Falkenroth 2000, p.172), there are systems with cyclic behaviour, in which non-termination is defined as correct, and consequently, the correctness of a rule set is application dependent. This means that rule set properties which are correct in one application may be incorrect in another application and vice versa.

2.3.2 Confluence

In a set of ECA rules which are triggered simultaneously, there might be dependencies between the rules, causing a non deterministic behaviour of the system. The reason for this is that the outcome of the execution of two simultaneously triggered rules may depend on in which order the condition part of the rules where evaluated, or in which order their action parts were executed (Paton & Diaz 1998). This is because the condition evaluation or action execution of rule R1 may affect the outcome of the condition evaluation or action execution of rule R2 and vice versa.

The termination and confluence properties are important characteristics for a set of active rules and they are given a lot of attention in the research area of active rules. There are different approaches to tackle these problems, Comai & Tanca (2003) are for example translating the rule set of any ADBMS into an internal language and logical clauses, proving whether the processing of the active rule set is deterministic or not. The work of Comai &
Tanca (2003) is taking the semantic differences in different execution models into account. These differences may cause a rule to behave differently depending on in which ADBMS it is executed.

2.3.3 A Distributed Active Real-time Database System (DeeDS)

The implementation of ECA rules varies in syntax and semantics. In some cases, the semantic of a rule is easy enough to be expressed in general terms, and then transformed to the exact syntax of the actual ADBMS. However, in cases where for example time is considered, and event occurrences are triggered relative to each other, it is beneficial to use an existing ADBMS as a framework for transformations. In this project, the ECA rules introduced in the rule manager of DeeDS (Andler, Berndtsson, Eftring, Eriksson, Hansson & Mellin 1995) will be used in cases where a specific framework is needed. The framework could be any commercial or academic ECA syntax supporting timeliness, however, the timeliness requirement are likely to require some extended form of ECA rules, which is specific for each database. In DeeDS, it is possible to specify temporal attributes like for example deadlines and execution times in the extended form of ECA rules proposed by Eriksson (1998).

Since predictability is an important characteristic in real-time systems, the only coupling mode allowed in DeeDS is detached. If the action is allowed to execute in the triggering transaction, the execution time of the triggering transaction will vary depending on if the action part of the rule is executed, and in that case, the execution time of the action. To be sure that the execution time of the transaction is not changed, the action part of the triggered rule is executed in a separate transaction (Eriksson 1998).
2.4 Formal methods

The software developed today is becoming more and more complex and is used in increasingly critical situations. The correctness of computer systems are becoming a dominant issue for a large class of applications, and one approach to gain higher confidence in a system is to use formal methods. However, it is important to understand that formal methods are not a silver bullet for developing perfect software. According to Hall (1990) their most fundamental limitations arise from two facts: some things can never be proven and we can make mistakes in the proofs of the things we can prove.

Since the real world is not a formal system, a proof does not show that things in the real world will happen as you expect. You can never be sure that your specifications are correct, no matter how much you prove about them. However, despite that the use of formal methods does not give any absolute guarantees for the system correctness, formal specifications are better at exposing mistakes made than informal specification methods. It is also the case that as an error is exposed; people are more ready to agree that it is an error than in an informal specification, where it is sometimes not clear what is being said (Hall 1990).

2.4.1 Proof of correctness

Although it is not possible to prove the correctness of the specification with respect to the real world, some things actually can be proven. Using a specification based on mathematics makes it possible to demonstrate that one formal statement follows from another and this is how a proof is conducted. However, even if there is a successful proof showing that the implementation corresponds to the specification, the behaviour of the system is affected by three factors limiting the usefulness of the proof: the behaviour of the programming language, the operating system and the underlying hardware (Hall 1990). These limitations
are crucially important for safety critical systems which imply that the proof must be complemented with systematic testing methods, especially for non-functional properties such as timeliness.

2.4.2 Benefits of formal methods

Any high quality system must meet both its functional and non-functional requirements. Even if it is possible to catch and correct errors and deviations from the specification in the last phases of a system development project, it might be very expensive to correct the errors in late stages of a development process. This implies that a fault avoidance technique, like formal methods, which have high probability of catching errors in the early stage of the development process, might reduce the total development cost due to less effort spent on testing and correcting expensive errors.

The process of showing that the system satisfies its specification is named system verification. To be able to perform a formal verification of a system, including formal proofs, a formal specification is needed (Wing 1990). However, since the specification itself might be wrong, the use of formal methods must be complemented with testing. The validation of a system is mainly concerned with the issue of testing and debugging. However, formal specifications can also be used in the testing phase, to generate test-cases, reducing the time spent on testing.

2.4.3 Formal schema for composite events

The state of a system which is implemented using active rules can be represented as histories of event occurrences. In such models, each update of an event history (each time an event occurs) causes a change of state. In Mellin & Andler (2002) and Mellin (2003) a formalized schema for event composition processing is presented. This schema is
used in this dissertation as a base for proving equivalence of the behaviour between timed automaton specifications and occurrences of composite events. The reason for choosing the schema presented in Mellin & Andler (2002) is that it takes the processing of events into account, and it is possible to compare behaviour of operators in different contexts.

The schema presented in Mellin & Andler (2002) is based on set theory where state transformations are based on event occurrences and operator rules. For each operator in each context, a rule is formally formulated based on the set of occurred events, and as the precondition of the operator rule is true, a composite event is raised. Each operator maintains a history of used and invalidated event occurrences for that operator, represented as sets. An event is used as the event occurrence has been used to contribute to a composite event, and it is invalidated as it can not be used to form a composite event occurrence. In chronicle context, a used event is a consumed event that can not be used to form a new composite event, while a used event in recent context may contribute to several other composite event occurrences.

The current state is the state of all event histories, so an occurrence of a monitored event causes a change in the event histories and hence, causes a change of state. The inference system presented by Mellin & Andler (2002), and in short described in this section and in section 5.3, is a rule based transformation between states.

Event occurrences

An event of type E may be primitive, denoted \( \text{prim}(E) \), or composite. Axioms stated for the event schema described in Mellin & Andler (2002) are that primitive event occurrences are instantaneous and there are no simultaneous occurrences of primitive events of the same type.

A composite event occurrence is composed by other event occurrences and may occur over a time span. The symbol \( \gamma \) is used to denote an occurrence of an event, where
CHAPTER 2. BACKGROUND

$\text{start}(\text{span}(\gamma))$ is the time of occurrence for the initiator event and $\text{end}(\text{span}(\gamma))$ is the time of occurrence for the terminator event of the event occurrence $\gamma$. The relation $\Gamma(E, [t, t'])$ is used to describe that an event of type $E$ has occurred in the time interval $[t, t']$.

The relation between initiators/terminators and the composite event they have initiated or terminated are described as an ordered pair $\langle \gamma', \gamma \rangle$, where $\text{iot}(\langle \gamma', \gamma \rangle)$ gives the set of $\gamma'$, where $\gamma'$ is either the initiator, or the terminator, of the composite event occurrence $\gamma$. The subscript $\omega$ is used to denote terminators and $\alpha$ is used to denote initiators.

Event types

The set of all monitored event types is denoted $E_m$. The set of event types that can initiate a composite event is denoted $E_\alpha$ while the set of event types that can terminate a composite event is denoted $E_\omega$.

Occurrence histories

The generated set $G$ contains the history of all primitive and composite events that have occurred so far. Each time a primitive or composite event occurs, it is inserted in the set $G$.

$$\forall \gamma \in G (\text{end}(\text{span}(\gamma)) \leq \text{now})$$

As an event is used or invalidated, it is inserted in the set of used or invalidated events for that event type; $U_{\alpha}(E)$ is a set of $\langle \gamma', \gamma \rangle$ for $E$ where $\gamma'$ is the used initiator; $U_{\omega}(E)$ is a set of $\langle \gamma', \gamma \rangle$ for $E$ where $\gamma'$ is the used terminator; $I_{\alpha}(E)$ is a set of invalidated initiator occurrences of type $E$ and $I_{\omega}(E)$ is a set of invalidated terminator occurrences of type $E$.

The tuple $(U_{\alpha}(E), U_{\omega}(E), I_{\alpha}(E), I_{\omega}(E))$ describes the state of each $E$.

Let $s_0, s_1, \ldots, s_n$ represent the history of states of event composition, then $s_i$ is $G$ and
the set of operator states. That is, \( s_i \) is defined as \( G, \cup_E (U_\alpha(E), U_\omega(E), I_\alpha(E), I_\omega(E)) \). In other words, the state \( S_i \) is the event history \( G \) together with all used and invalidated event histories for all monitored event types. Each operator type has a set of operator rules that transforms \( s_i \) into \( s_j \).

**Example**

Assume \( G = \{ \Gamma(E_1, [1, 1]), \Gamma(E_1, [2, 2]), \Gamma(E_2, [3, 3]), \Gamma(E_2, [4, 4]), \Gamma(E_2, [5, 5]) \} \)

The composite event type \( E = E_1; E_2 \) is monitored in chronicle context, the set of monitored event types \( E_m \) is \{ \( E, E_1, E_2 \) \}, the set of initiator types \( E_\alpha \) for \( E \) is \{ \( E_1 \) \} and the set of terminator types \( E_\omega \) for \( E \) is \{ \( E_2 \) \}. Given the occurrence history \( G \), two occurrences of \( E \) will be formed using the following event occurrences: \{ \( \Gamma(E_1, [1, 1]); \Gamma(E_2, [3, 3]) \) \} and \{ \( \Gamma(E_1, [2, 2]); \Gamma(E_2, [4, 4]) \) \}. The sets containing used and invalidated occurrences will be the following:

\[
U_\alpha(E) = \{ \langle \Gamma(E_1, [1, 1]), \Gamma(E, [1, 3]) \rangle, \langle \Gamma(E_1, [2, 2]), \Gamma(E, [2, 4]) \rangle \}
\]

\[
U_\omega(E) = \{ \langle \Gamma(E_2, [3, 3]), \Gamma(E, [1, 3]) \rangle, \langle \Gamma(E_2, [4, 4]), \Gamma(E, [2, 4]) \rangle \}
\]

\[
I_\alpha(E) = \emptyset
\]

\[
I_\omega(E) = \{ \Gamma(E_2, [5, 5]) \}
\]

In other words, the event occurrences that have initiated a composite event of type \( E \) is included in the set of used initiators for that type \( (U_\alpha) \), and the event occurrences that have terminated the composite event \( E \) is included in the set of used terminators for that type \( (U_\omega) \). In the example history \( G \) there are no invalidated initiator occurrences, however, the event occurrence \( \Gamma(E_2, [5, 5]) \) is an invalidated terminator occurrence, since there are no matching unused occurrence of the initiator type \( E_1 \).
2.4.4 Finite-state machines and Finite automata

The behaviour of many kinds of machines, including components in computers, can be modelled using a structure called finite state machine. A state machine consists of a set of states, including a starting state, a set of transitions, an input alphabet, a transition function that assigns a new state for each pair of state and input, and an output function that assigns an output to each pair of states and inputs (Rosen 1999).

A model that is closely related to finite-state machines is finite automata. The difference is that finite automata, instead of producing an output sequence, have a set of final states. The finite automaton produces an acceptance or rejection of the input sequence, depending on if a final state is reached by the input sequence or not. Finite state automata can be used as language recognizers, which is a preferable characteristic in modelling compilers (Rosen 1999). A string is said to be recognized (or accepted) by a finite automaton if it takes the machine from the initial state to a final state. The language recognized (or accepted) by a state machine is the set of all strings that are recognized by the state machine. Two finite state automata are called equivalent if they recognize the same language (Rosen 1999).

Finite-state automaton example

As previously exemplified in Ericsson (2002), a finite-state automaton can be described as a tuple \((\Sigma, S, S_0, E, F)\), where \(\Sigma\) is the input alphabet, \(S\) is a set of states, \(S_0\) is the initial state and \(E\) is a set of edges (\(E\) is a subset of \(S\times S\times \Sigma\)). Given the input \(\alpha\) the state of the automaton is transforming from \(s\) to \(s'\) if \(\{s, s', \alpha\}\) is a subset of \(E\) (Alur & Dill 1994). \(F\) denotes the set of states accepted by the automaton. The corresponding values for the finite-state automaton in Figure 2.4 are the following:
Figure 2.4: State diagram of a finite-state automaton

- $\Sigma = \{1, 0\}$
- $S = \{S_0, S_1, S_2\}$
- $S_0$ is the starting state.
- $E = \{(S_0, S_0, 0), (S_0, S_1, 1), (S_1, S_0, 0), (S_1, S_2, 1), (S_2, S_1, 0)\}$
- $F = \{S_0, S_2\}$

A word $\alpha$ over the alphabet $\Sigma$ in the example may be 001011. The language $L$ recognized by the automaton $M$ in Figure 2.4 is the sets of inputs that take the automaton from its start state $S_0$ to one of the final states $S_0$ or $S_2$. The set of strings accepted by the automaton is denoted $L(M)$. To reach the final state $S_0$ or $S_2$, any string of ones and zeros that are not containing more than two consecutive ones, and that does not contain a single one, are required.

The finite state automaton described in Figure 2.4 is a deterministic automaton. In a deterministic automaton there is a unique next state given by the transition function for each state and input pair. If there can be several possible next states for each pair of state and input value, the automaton is said to be nondeterministic (Rosen 1999).
Finite automata and regular languages

A language of an automaton is every string that is derivable from the starting state. Every language that is accepted by a finite automata is a regular language and every regular language is accepted by a finite automaton (Sallings 1998, p. 43). This result was proved by Kleene 1956 and is called Kleenes theorem.

A regular language can be expressed in a set notation called regular expressions (Sallings 1998, p. 23). The operators of regular expressions are Kleene closure, union and concatenation. A concatenation of a set A and B, denoted AB, is the set of all strings of the form xy where x is a string in A and y is a string in B. The Kleene closure of a set A, denoted by \( A^* \) is the set consisting of concatenations of arbitrary many strings from A (Rosen 1999, p. 648).

If \( \{0\} \cup (1\{0\}) \) is a regular language over the alphabet \{1,0\}, then the corresponding regular expression is \( (0 \cup 1\{0\}^*) \). The automaton accepting the language \( \{0\} \cup (1\{0\}^*) \) is shown in Figure 2.5. To reduce the number of parentheses used, there are order conventions to use. The order conventions of regular expressions are first Kleene closure then concatenation and last the union operator. The expression \( (0 \cup 1\{0\}^*) \) can then be written \( 0 \cup 1\{0\}^* \) (Sallings 1998). The notation \( A^+ \) is equal to \( A^* \) except that \( A^+ \) does not contain the empty string. Table 2.1 shows some examples of regular expressions.

In cases where it is interesting to know whether a language of one finite automaton
CHAPTER 2. BACKGROUND

<table>
<thead>
<tr>
<th>Expression</th>
<th>Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^*$</td>
<td>A 1 followed by any number of 0s (including no zeros)</td>
</tr>
<tr>
<td>$(10)^*$</td>
<td>Any number of copies of 10 (including the null string)</td>
</tr>
<tr>
<td>$0 \cup 01$</td>
<td>The string 0 or the string 01</td>
</tr>
<tr>
<td>$0(0 \cup 1)^*$</td>
<td>any string beginning with 0</td>
</tr>
<tr>
<td>$(0^<em>1)^</em>$</td>
<td>Any string not ending with 0</td>
</tr>
</tbody>
</table>

Table 2.1: Some regular expressions rewritten from (Rosen 1999, p. 657)

is a subset of the language recognized by another finite automaton, this problem can be seen as a string comparison problem, which checks whether two strings are identical (Brookshear 1989, p. 269). The expressive power of a regular language is limited due to its finite memory and inability to count inputs and outputs. This is why a more expressive language is needed in more complex situations.

2.4.5 Timed automaton

Finite state machines can not reason easily about time, since in the best case, time must be represented by counting clock ticks, which may cause a state explosion. Instead, to include time constrains in the finite automaton model, a timed automaton may be used. A timed automaton is a finite automaton extended with a set of real-valued clocks. A timed automaton accepts timed words, which means that a real valued time of occurrence is associated with each symbol. As an automaton makes a choice to take a state transition, the choice of the next state depends both on the input symbol read and the time value of the input symbol.

The clocks may be reset independently of each other during a transition and their value may be used as guards on transitions. The addition of a finite set of real value clocks makes it possible to prove real-time requirements of finite-state systems (Alur & Dill 1994).

The state of the timed automaton is the state of the finite automaton together with a clock value. A set of clocks is added to the transition table forming a timed transition
TABLE 2. BACKGROUND

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{state_diagram.png}
\caption{State diagram of a timed automaton}
\end{figure}

According to Alur & Dill (1994) a timed transition table are defined as a tuple \((\Sigma, S, S_0, C, E)\) where \(\Sigma\), \(S\) and \(S_0\) are defined as in a regular automaton, \(C\) is a finite set of clocks and \(E\) gives the set of transitions. The set of transitions in a timed automaton is increased with the set of clocks to be reset in the transition, and the set of clock constraints for the transition. Each symbol presented to the automaton represents an occurring event, and the time value attached to it represents the time when the symbol is presented to the system. According to Alur & Dill (1994) a pair \((\sigma, \tau)\) is referred to as a timed word over the alphabet \(\Sigma\) if \(\sigma\) is an infinite word over \(\Sigma\) and \(\tau\) is an increasing time sequence. A set of timed words over \(\Sigma\) is referred to as a timed language.

**Timed automaton example**

The example automaton in Figure 2.6 has the alphabet \(\Sigma = \{0, 1\}\). An accepted language \(\mathcal{L}\) for this automaton would be a sequence of zeros while time is below 5, followed by a single one as the time increases above 5, followed by an arbitrary number of zeros.

A formal notation is often used to express the language of an automaton. The formal notation for the accepted language in the example would be:

\[
\mathcal{L} = \{(\sigma, \tau) \mid \forall i(\tau_i < 5 \rightarrow \sigma_i = 0) \land \exists j((\tau_j \geq 5 \land \forall i(\tau_i < \tau_j \land \sigma_i = 0)) \rightarrow \sigma_j = 1) \\
\land \forall i((\tau_i \geq 5 \land \exists j(j < i \land \sigma_j = 1)) \rightarrow \sigma_i = 0)\}
\]
In the example \( \forall i (\tau_i < 5 \rightarrow \sigma_i = 0) \) express the acceptance criteria of only zeros as the time is below 5, \( \exists j ((\tau_j \geq 5 \land \forall i (\tau_i < \tau_j \land \sigma_i = 0)) \rightarrow \sigma_j = 1) \) captures the criteria of a one if the time is above or equal to 5 and all previous inputs are zero and \( \forall i ((\tau_i \geq 5 \land \exists j (j < i \land \sigma_j = 1)) \rightarrow \sigma_i = 0) \) denotes that after the single one there will be an arbitrary number of zeros.

Timed regular languages are defined by Alur & Dill (1994) to be the class of timed languages accepted by a timed Büchi automaton. Such automata have a set of accepting states and a run over the automaton is accepting if some accepting state is repeated infinitely often. This is for example the case in Figure 2.7 which accepts the timed language \( \mathcal{L} = \{((ab)^\omega, \tau) \mid \exists i \forall j \geq i (\tau_{2j} < \tau_{2j-1} + 2) \}. \)

### 2.4.6 Uppaal

A timed automaton is analysable; it is for example possible to check if a certain state is reachable within a limited time period. However, performing these checks manually is difficult and time consuming. Fortunately there are CASE tools developed with the aim of solving these tasks. Uppaal is a CASE tool developed jointly by Uppsala University and Aalborg University. The analysing capabilities of Uppaal are for example model checking and reachability analysis (Larsen et al. 1997). A model is built up by one, or more timed
CHAPTER 2. BACKGROUND

automata. Each timed automata is simulating a process which is able to synchronize with other automatons. As the model is built, it is possible to visually simulate its behaviour in the CASE tool, as well as verifying its correctness using queries.

Both liveness and safety properties may be checked using the verifier in Uppaal. Checking safety is to ensure that bad things never happen. Consider for example a railway control system, which must guarantee that at most one train can pass some critical point at a time. If the railway control is modelled in Uppaal, it is possible to automatically check if this property holds using the verifier. It is also possible to check that good things eventually happen (liveness), imagine for example that a certain train must be able to cross a critical track section within a specific time period (Yi, Pettersson & Daniels 1994).

The notation for an initial state in Uppaal is a double circle, but there is no explicit notation for an accepting state. In this dissertation, all figures showing an automaton is produced by Uppaal, so accepting states are named \(<\text{State name}>\) accepting, since there is no special notation for it in Uppaal. Since the start state in Uppaal has the same notation as an accepting state in other notations (double circle), the start states are named \(<\text{State name}>\) start in this dissertation to avoid confusion.

Since a timed automaton specification can be built up by a network of timed automata, there is a need for communication between cooperating automata. In Uppaal this communication is performed by synchronizing on global channels. If an automaton sends information on channel x (the exclamation mark x! is the notation for send), another automaton must be able to synchronize and receive the message (the question mark x? is the notation for receiving a message on the channel x). If two automatons are synchronizing on a channel, they may only take the synchronising transition if the synchronizing automaton takes its transition simultaneously.

Figure 2.8 shows an example of a fragment of an Uppaal specification with two synchronising timed automata. The first automaton consists of state P1, P2 and P3 and the
second automata consist of the state S1, S2 and S3. In state S1, there is a time invariant $x \leq 4$, meaning that the system may not remain in this state as the time is above or equal to 4, and on the transition between S1 and S2 there is a time guard $t == 4$. The invariant together with the time guard forces the timed automaton to take the transition between S1 and S2 exactly as the clock variable $t$ is equal to 4. In state S2, the system is not allowed to wait for more than one time tick, since there is an invariant for $t \leq 5$.

The name convention for variables in timed automata models in this dissertation is that clock variables starts with t, guards are placed at the very start of transitions, synchronisations are placed on the middle and assignments are placed at the end of transitions.

In this report, there are figures produced in Uppaal which are not exactly following the Uppaal semantic. The reason for this is to gain increased understandability of the issues described. It is for example not yet possible to specify a guard as true or false in Uppaal, as in Figure 5.2, an integer which may take one of the values one or zero are usually used for this purpose.

Uppaal is not the only CASE tool available for specifying systems in timed automata. Two other tools worth mentioning are Kronos and HyTech (Yovine 1997). The CASE tools have both advantages and disadvantages against each other, both in the modelling
and analysing capabilities (Berard & Sierra 2000). However, the ability to model integers in Uppaal, which is useful when it comes to modelling composite events, is one of the reasons why Uppaal is chosen for this project.

2.5 Timed automata to ECA rules

In a recent study, Ericsson (2002) proposes a method for deriving ECA rules from timed automata specifications. The main idea of this method is to transform the analysed behaviour of a timed automaton into a set of implicitly analysed rules. The specifications are considered to have equivalent behaviour if they are producing an identical output sequence given an identical sequence of inputs (Abadi & Lamport 1991).

In Ericsson (2002) a case study was performed, where the method was applied on a timed automaton specification of a production cell. The outcome of applying the method was a set of rules which maintained the behaviour of the timed automaton, given that the order of event occurrences was predictable.

A prerequisite for the developed method is to divide the set of states in the automaton specification into external and internal states. External states are states which are affecting the systems external behaviour and internal states are defined as states which are only serving as help states needed in the specification language, for example to model time delays.

The method is based on that each arrival to an external state in the timed automaton is transformed to an event occurrence in the rule base. Transition assignments are mapped to methods, which are executed in the action part of the derived ECA rule.

The decision to map each arrival to an external state in the automaton to an event occurrence in the rule base has several advantages, however, it also causes a mismatch in the behaviour of an automaton and a rule set if there is a guard on the transition. It is not
possible to map the guard in the automaton to a condition in the rule set for all cases. This is because an automaton is waiting in its state until a guard becomes true, but an ECA rule evaluates its condition once, and if the condition is false, the rule condition will not be evaluated again until a new event of that type occurs. In the previous case-study, this mismatch was solved by using composite events; however, a general solution for guarded transitions is missing.

Rule derivation approach

The method for deriving ECA rules assumes that the specification is modelled in Uppaal and built up by several synchronizing sub automata. The method developed in Ericsson (2002) consists of the following steps.

I Identify external states.

Determine which states in each automaton are internal (help states for the automaton), and which states are external, and relevant for the forthcoming derivation of rules.

II Identify preconditions for external transitions.

For every transition between two external states in every model, determine which states all sub automata must be in for the transition to be executed (the event acceptance state of the entire model), and values of relevant variables.

III Identify post conditions for external transitions.

For every transition between two external states in every automaton, determine which states all sub automata must be in and values of relevant variables immediately after the transitions.
IV Express the action part of the transition as a method where it is possible.

V Identify redundant transitions.

Determine if redundant transitions, e.g. transitions that have the same pre- and post- conditions, and that are synchronising, can be composed into one single rule.

VI Express the entering of external states as primitive event occurrences.

Express the entering of every external state in the model as a primitive event in the ECA rule specification language.

VII Identify and specify composite events.

Identify if there are composite events (If the precondition of the rule depends on more than one automaton), and in that case, which primitive events the composite event are combined of.

VIII Formulate the set of rules.

Attach the condition and action parts of the transition to formulate rules

A proof of principle was performed as a case study where a timed automaton specification of a production cell for treating metal bricks were transformed into a set of ECA rules. The case study was focused on the correctness of the behaviour of the entire application, and did not consider the general case.
Chapter 3

Problem description

This section will clarify the aim of this project, as well as present the arguments behind the aim.

3.1 Active real-time database applications

Event triggered real-time systems may take advantage of the persistent storage and event monitoring capabilities of an active database. However, despite lots of prior work in the area of active databases, ECA rules are still notorious for its low level specification language, which makes such systems really hard to analyse and maintain. The rules are affecting each other, directly and indirectly, in a way that makes a thorough analyse of the systems behaviour impossible if no constraints are set on the design of the rule set (Falkenroth 2000). In contrary, the desire to use active databases in real-time systems requires an analysed and predictable behaviour of the rule set, which also must guarantee that certain deadlines can be met.

The problem of analysing the behaviour of a set of rules, and the desire to use active databases in real-time systems, highlights the need for a method which facilitates the
development of predictable active real-time applications. According to Paton & Diaz (1998) the area of developing active applications lack appropriate design methodologies, even in cases where time constraints are not considered. Several approaches exist for verifying different characteristics of an existing rule set, however, the development of the set of rules and the verification of the behaviour of the entire rule set in a certain execution model with respect to its specification, are usually not considered. The set of rules are usually assumed to magically appear and there is poor guidance for the developer on how to create a predictable rule set from for example a formal specification, and how to design a set of rules which is possible to maintain, change and reuse.

3.2 Deriving active rules from timed automata

There are several benefits of specifying and analysing the behaviour of a real-time system in a formal timed automaton notation. It is for example possible to verify that a certain state will be reached within a specific time, and that the system will not deadlock (if the system implements the specification correctly). The correctness of the models may be automatically analysed by some of the CASE tools available for automatic verification and simulation of timed automaton models, for example Uppaal (Larsen et al. 1997).

However, there is no obvious way of taking advantage of these benefits when specifying a real-time application using an active database. Furthermore, there is no CASE-tool or method support for verifying that the behaviour of the implemented rule set corresponds to the behaviour specified in the timed automaton. In the ultimate scenario, the set of rules that implements the behaviour of the timed automaton is automatically generated from the timed automaton specification.

To reach the vision of automatically generating active rules from an arbitrary timed automaton specification, a method is needed which transforms the exact semantics of the
CHAPTER 3. PROBLEM DESCRIPTION

timed automaton into a set of ECA rules. This means that as an event occurs, it must have the same effect in both the timed automaton and the rule set. However, it is not enough that the rule set is reacting on all event occurrences that are specified in the timed automaton, it is also the case that in a given state, the rule set may only react on event occurrences that is also accepted in the timed automaton in that particular state. As the exact behaviour of the timed automaton is transformed to a set of ECA rules, the behaviour of the rule set is implicitly analysed, since the analysed behaviour of the timed automaton is maintained during the transformation.

In the work of Ericsson (2002), a method was proposed for transforming a timed automaton specification to ECA rules. A proof of principle was performed as a case study, however, the transformations were application specific, and transformations from arbitrary specifications was not considered.

Recall that an active database may react on both primitive and composite event occurrences. A primitive event occurrence is defined by Ericsson (2002) to be transformed from the arrival to an external state in the timed automaton. However, the semantics of for example synchronization is transformed to a composite event, and in such cases the transformation must also consider the current context. This is because the same set of states and transitions in a timed automaton may be transformed to different types of composite events depending on the current context.

To gain increased confidence in the proposed method, the equivalence of mappings between constructs in the timed automaton and the rule set must be further investigated and verified, especially with emphasis of composite events in different context, which was not addressed in Ericsson (2002).
3.3 Aim and objectives

The aim of creating an entire CASE tool for generating a rule set from a timed automaton is out of scope of this project. However, verifying the correctness of the transformation in some execution models and identifying patterns in the timed automaton specification, which are exactly mapping to occurrences of composite events in the rule set, is a step forward towards such automation. In this project, a subset of the transformation constructs identified in Ericsson (2002) will be reconstructed to an application independent format before they are transformed into ECA rules, to increase the applicability of the proposed method on arbitrary specifications. The transformed subset will be expanded to also include transformations of the behaviour between patterns of timed automata and composite events in different contexts.

The aim of this project is to verify correctness of transformations between a set of application independent constructs in timed automata and its resulting rules. The verified transformations will be a subset of the transformations performed in Ericsson (2002) expanded with transformations of composite events.

To reach the aim the following objectives must be fulfilled:

Generalize transformations and identify limitations of transformations

In the previous case study, the transformations were only verified for one particular application. To be able to apply the method proposed in Ericsson (2002) on an arbitrary specification, a subset of the transformations found in the previous case-study will be reconstructed to application independent constructs of timed automata before they are transformed into rules. In this way, the transformations are “generalized”, since small
constructs of timed automata are identified which will always be transformed to a certain rule. It will be verified that the generalized constructs maintain the specified behaviour during their transformation into rules.

It is also important to identify constructs in the timed automaton which are hard, or impossible to transform to rules using the proposed method.

**Specify behaviour of composite events in timed automata**

The previous case study did not address applications with unpredictable occurrence order of events, and since this is a common scenario in event triggered systems, it is interesting to expand the set of verified transformations with timed automata constructs that can accept unpredictable event occurrences. Specifying the behaviour of composite events in timed automata makes it possible to react on an equal set of unpredictable event occurrences in the timed automaton specification as is possible in the rule set.

The behaviour of composite events will be specified in one automaton for each operator as “operator patterns”. Besides facilitating the process of specifying event triggered systems in timed automata, the patterns may be used in the process of transforming an existing timed automaton specification to rules. The acceptance language for the patterns will be identified to facilitate the detection of operator patterns in arbitrary specifications.

The idea is that if an operator pattern is identified in an arbitrary timed automaton specification, this part of the automaton can safely be transformed to its corresponding composite event.

**Formally verify equivalence of semantics**

The equivalence of the behaviour between timed automaton patterns for conjunction, disjunction and sequence in chronicle context and their corresponding composite event operators will be verified.
CHAPTER 3. PROBLEM DESCRIPTION

It must be verified that the transformation between the timed automaton and its corresponding composite event maintains the exact behaviour of the timed automaton. The behaviour of an operator pattern is considered to be equivalent with its corresponding composite event if the specifications generate the same sequence of composite events for a given sequence of event occurrences.

3.4 Limitation of project scope

Since the semantic of an ECA rule may be different depending on the execution model, the execution model must be considered when deriving rules from a higher level specification. The number of different combinations of consumption policies (context), coupling modes and operators are huge, so this dissertation will only consider the combinations identified to be most interesting for real-time systems. The transformations will be verified for chronicle and recent context, since these contexts meets the requirements of real-time systems concerning temporal validity and order of occurrence. The formal proof section will only consider chronicle context since chronicle is more complex than recent context due to its order restriction and consumption of both initiator and terminator events.

The only coupling mode considered will be detached since this is the only coupling mode allowed in DeeDS (Eriksson 1998) and the operators used will be \{ or, and, sequence, expiration \}. The selection of the set of operators are based on what operators is supported in DeeDS and a survey performed by Paton & Diaz (1998) where the current support for different operators in a set of well known active databases are reported.
Chapter 4

Method

The method for deriving ECA rules from timed automaton specifications developed in Ericsson (2002) was previously proven successful in a case study. However, one successfully performed case study does not say much about the methods applicability in the general case. To gain more confidence about deriving rules from timed automaton specifications in the general case, it is necessary to verify the correctness of each transformation; each derived composite event, as well as the correctness of the derived rule set as a whole. The following sections present the methods that will be used to fulfil the objectives specified in section 3.3.

4.1 Generalize transformations and identify limitations of transformations

To verify the correctness of the method proposed in Ericsson (2002) for the general case, the semantics of identified constructs in the timed automaton specification must be mapped to ECA rules which maintain the behaviour of the timed automaton.

In most cases when specifying a system, there is knowledge about the system that
can help the developer to reduce the complexity of the specification. An example of such knowledge is the statement: “The conveyor belt is always in running mode before the sensor can be triggered”. Such knowledge may also simplify the transformation into rules, since the occurrence order of the events are known. In this objective, the order in which event occurs is assumed to be known, but no other a prior knowledge about the system to be specified is assumed, since the transformations will be verified for the general case.

For each timed automaton pattern considered in this objective, the verification of the proposed methods correctness in the general case will be performed in following steps:

I Firstly, constructs of timed automata found in the previous case study will be reconstructed to a format which is application independent and in that sense more general. The ECA rule primitives for expressing the semantics of the timed automaton constructs will then be identified.

II Secondly, the similarity of semantics between the timed automatons constructs and the ECA rule will be verified for the general case. The automaton constructs are expected to be simple enough for an informal reasoning to be a sufficient verification technique.

III If the timed automaton construct is not possible to transform for the general case, using the proposed method, this will be identified as a limitation and an alternative solution will be proposed where it is possible.

4.2 Specify composite events in timed automata

The behaviour of composite event occurrences is specified in separate timed automata, this makes it possible to react to sporadic event occurrences arriving in an arbitrary order in
the timed automaton specification. To reach the objective of specifying different types of composite event patterns in timed automata the following steps will be performed:

I Firstly, the behaviour of each selected composite event will be specified in timed automata in recent and chronicle context, with no focus on possible event parameters. In this step the modelling is only concerned about specifying the way that the event occurrences are formed into composite events in different contexts.

II Secondly, the events are assumed to carry important event parameters. This means that the complexity is increasing since the event parameters must be stored and associated to the correct event occurrences.

III As a third step, the acceptance language of the automaton pattern will be expressed. This is an important step since automatons with equal semantics may be differently specified. If the acceptance languages of two different automatons are equal, then we know that the two automatons are equivalent (Salling 1998). It will also be investigated if the timed automata pattern, describing the composition of events, is a regular expression or not. If the language accepted by the pattern can be described as a regular expression, it will be easier to find the pattern in an arbitrary finite automaton as discussed in section 2.4.4.

IV In chronicle context, the method for specifying composite event patterns contains a fourth step where expiration times are considered. If all expiration times are unlimited, the event monitor may be busy composing events which have already missed their deadlines. This is especially cumbersome in chronicle context, since the oldest unused event occurrences are used to compose events in this context.
4.3 Formally verify equivalence of semantics

It will be verified that each set of event occurrences is producing identical composite events as output in both the automaton and the rule set. The specifications are assumed to have identical behaviours if they produce identical output given the same sequence of input events (Abadi & Lamport 1991). The output is recorded in event histories, and the verification is successful if it can be proven that the event histories of the automaton and the active rules are identical for the same sequence of input events. The verification process will be performed in the following steps:

I Present the operator predicate of the composite event in text and formally.

   It is important to use a formal approach for describing the composite event operator, since this will provide a solid base for the formal proof. A formal event processing schema presented in Mellin & Andler (2002) will be used where formal rules are defined for each operator. As a first step the operator rules defined in Mellin (2003) will be presented and explained since they will be used as a reference for the behaviour of the composite event in the forthcoming proof.

II Formalize important variables of the timed automaton pattern.

   If the timed automaton pattern depends on variables for its correctness (for example counters), these variables must be formalised since they must be included in the proof.

III Express the invariant of each timed automaton state.

   The state of the system is defined in the histories of occurred, used and invalidated event occurrences. Describing states in event histories makes it possible to compare the state of the rule base with the state of the timed automaton pattern. For each
state in the timed automaton, an invariant is described in the set of occurred, used
and invalidated events.

IV Show that all possible inputs in each state correspond to the invariant of the next
state, and show that each signalling of a composite event in the automaton corre-
sponds to a signalling of a composite event in the rule base.

This step is performed as a truth table, where the effect of each possible input in
each state is recorded. If there is a change of state in the timed automaton due to
the event occurrence, the defined invariant of the new state must be satisfied, if the
timed automaton remains in the same state, the invariant of the current state must
remain satisfied after the event occurrence.
Chapter 5

Verify transformations

In the following sections the result of applying the methods described for each objective is reported.

5.1 Generalize transformations and identify limitations of transformations

The method proposed by Ericsson (2002) is built on three basic assumptions. The first assumption is that an arrival to an external state in the timed automaton can be transformed to an event occurrence in the rule base. The second assumption is that assignments in a timed automaton transition can be transformed to methods, which implements the assignments, in the action part of a rule. The third assumption is that as an action method is executed, a new state is reached, and a new event is raised.

In the following section, an overview of general transformations of different sub parts of timed automata (for example arrival to a state, guard and time delay) is first presented. As a second step, identified limitations of transformations for the general case are reported. The environment is initially assumed to be predictable in the sense that the occurrence
order of the events is known a priori. An example of such systems is a manufacturing plant where events occur in a predefined order.

5.1.1 Enter state

According to the method proposed in Ericsson (2002), each arrival to a new state in the timed automaton is transformed to a primitive event occurrence in the rule set, if the state is defined to be external. An external state is a state which affects the external behaviour of the system, internal states are help states whose only purpose is to facilitate the modelling of the timed automaton, for example to model the time it takes to perform a transition.

The state of a rule set may be expressed in histories of occurred, used and invalidated events. Mapping the arrival to an external state in a timed automaton to an event occurrence implies that a change of state occurs simultaneously in the rule set and in the timed automaton.

In this dissertation, if nothing else is defined, the arrival to an external state is transformed to a primitive event named $E_{\text{arriveTo}\langle\text{Statenone}\rangle}$. As a timed automaton for example arrives to state S1, this happening is corresponding to the occurrence of event $E_{\text{arriveToS1}}$ in the rule base.

A semantic difference between a timed automaton and a set of rules is that as a timed automaton arrives to a state, it may stay for an arbitrary long time in that state before it evaluates a guard or takes the next transition. This is not true in the rule set where the evaluation of conditions and execution of actions are controlled by the current coupling mode. To force the automaton to behave more as a rule, the facility to mark a state as urgent in Uppaal is used, which forces the automaton to leave the state as soon as possible. In Figure 5.1 a simple automaton is modelled where state S1 marked as urgent.
5.1.2 Transition assignments

As a timed automaton takes a transition, it is possible to assign a value to a variable or a clock. This assignment may for example open a valve to decrease the pressure in a tank, or signal that a robot arm shall release an object.

As the timed automaton is transformed into rules, these assignments are transformed into methods which are executed in the action part of the ECA rule. As the action method is executed (the assignments are performed in the timed automata), a new event is raised in the rule base, which is corresponding to the arrival of a new state in the timed automaton as the transition is taken.

If nothing else is defined, the assignment part of a transition is transformed into the method action < Startstate > To < Endstate > (). As for example the assignment part of the transition between state S1 and state S2 is transformed, the action part of the rule is to execute the method actionS1toS2().

Since the target platform is DeeDS (Andler et al. 1995), the action part of the coupling mode is detached, meaning that the action part of the rule is executing in a transaction on its own. This behaviour is well mapped to a timed automaton since a transition is similar to a transaction in that it executes entirely or not at all, and it is executing as one single unit.
5.1.3 Guards

If the timed automaton specification includes guards, the guards may be transformed to the condition part of the derived ECA rule. However, this transformation is only preserving the specified behaviour under the following conditions:

I Each state with guarded transitions has exactly one valid\(^1\) transition at each point in time.

II The occurrence orders of the events are deterministic, i.e. the events occur in the order specified in the timed automaton.

This is for example true for the automaton in Figure 5.2 where the guard is either true or false, and the automaton has a transition for both cases. The cases where there are none, or more than one possible transition to take from a state are discussed in section 5.1.6. The following rules are specifying the same behaviour as the automaton in Figure 5.2, provided that the identified preconditions are true.

Rule S0toS1:
ON $E_{\text{arrivalToS0}}$
DO actionS0toS1()

Rule S1toS2:
ON $E_{\text{arrivalToS1}}$
IF guard == true
DO actionS1toS2()

\(^1\)Valid in this case means that the transition is possible to take. Its guard is true, and if the transitions is synchronised, it is also possible to synchronise with another transition at that moment.
CHAPTER 5. VERIFY TRANSFORMATIONS

![Diagram of a simple automaton with urgent states and guards]

Figure 5.2: Simple automaton with urgent states and guards

Rule S1toS3:

ON $E_{\text{arriveToS1}}$

IF guard == false

DO actionS1toS3()

The transition between S0 and S1 in the automaton corresponds to Rule S0toS1. The action part of this rule raises an event occurrence of type $E_{\text{arriveToS1}}$ that triggers both Rule S1toS2 and Rule S1toS2. Both of the rules will evaluate their condition parts, but only one of the rules are executed, since only one of the conditions $\text{guard == true}$ and $\text{guard == false}$ is true as the system is in state S1.

If the evaluation of condition, or the execution of the action part of one of the rules, may change the outcome of the evaluation of the condition evaluation of the other rule, the execution model must make sure that only the rule which condition was true as the event $E_{\text{arriveToS1}}$ occurred is executed.

52
CHAPTER 5. VERIFY TRANSFORMATIONS

5.1.4 Time constraints

In timed automata it is possible to specify time constraints and the time is represented by a set of real-value clocks. In Uppaal it is possible to reset a clock during a transition and its value may serve as a guard on a transition, or an invariant in a state. By combining these features, a number of time constraints can be modelled in a timed automaton.

In the rule base it is possible to raise an event occurrence at an absolute point in time (e.g. the 13th of November 1998 at 15:00), the time relative to something else (e.g. 10 days after the shares are sold), and periodic time events (e.g. 8:00 every day) (Paton & Diaz 1998). The target database in this study is assumed to have the same time support as specified for DeeDS (Andler et al. 1995). In DeeDS it is possible to specify event occurrences for absolute time as well as for time relative to other event occurrences, periodic event occurrences, expiration times for events and deadlines for a rule (Eriksson 1998).

Time delay

Figure 5.3 shows how a clock guard can be used together with a time-invariant in a timed automaton to model a time delay. The clock $t_1$ is reset during the transition between S0 and S1. In state S1 the invariant $t_1 \leq 10$ implies that the system must not remain in this state if the value of the clock $t_1$ is above 10. The guard $t_1 == 10$ on the transition between S1 and S2 together with the invariant is forcing the system to take the transition between S1 and S2 as $t_1$ is equal to 10.

In DeeDS, the time of occurrence is denoted $tocc$, and a new event can be specified to occur after a time relative to this event occurrence (Eriksson 1998). To transform the delay shown in Figure 5.3, it is possible to specify a relative time event:

$$\textbf{event } E_{\text{TimeDelayS1}} \textbf{ is after } E_{\text{ArriveToS1}}[tocc(E_{\text{ArriveToS1}} + 10)]$$
Figure 5.3: Modelling delay of 10 time units in state S1

An event of type $E_{Time\_DelayS1}$ is raised 10 time units after the occurrence of an event of type $E_{ArriveToS1}$. The occurrence of the event of type $E_{Time\_DelayS1}$ triggers the rule that takes the system to state S2 after a delay of 10 time units.

**Deadline**

In Uppaal, the system can be verified to meet its deadline through reachability analysis. It is for example possible to ask queries to the system which answers whether a certain state will be reached within a certain time. For such constraints to be transformed into the rule set, each action method must have a specified worst case execution time.

In DeeDS it is possible to express that the action must have been executed within a certain time relative to the event occurrence, as shown in the following example where the system must reach state S1 within 10 time units after the occurrence of an event of type $E_{ArriveToS1}$:

```text
ON $E_{ArriveToS1}$
DO within tocc + 10 criticality soft
begin
    ActionS0toS1();
end
```
CHAPTER 5. VERIFY TRANSFORMATIONS

In DeeDS, the keyword `criticality` specifies information to the scheduler. The criticality can be soft, firm, hard essential or hard critical. Depending on the specified criticality, the execution time is treated by the scheduler to be the worst case execution time or average execution time (Eriksson 1998).

Clock guards

The necessary preconditions previously identified for guards based on variables (exactly one available transition to take, and a primitive event occurrence triggering the rule) are also necessary for clock guards. The automaton in Figure 5.4 represents a system where the time it takes to transfer from state S1 to S2 affects the choice of the next transition. As the transition between S0 and S1 is taken, the clock $t_1$ is reset and if the time it takes to transfer from state S1 to state S2 exceeds 4 time units, the transition between S2 and S4 is taken, else the transition between S2 and S3 is taken. In this example, the arrival to state S2 is simulating that a milestone of the execution is reached, where it is possible to decide whether a task runs the risk of missing a deadline. This may for example represent a real-time system which must execute a contingency action if the system is about to miss a deadline. If the system is right on time in state S2, the transition between S2 and S3 will be taken, however, if the system is late, the transition between S2 and S4 will be taken instead, executing procedures that produces results of lesser quality but in a timely manner.

As the automaton in Figure 5.4 is transformed into rules, the outcome clearly depends on the expressiveness of the rule language in the target database. In this case, it is assumed that events can be specified using relative validity as in the following example where the occurrence of primitive events of type $E_{\text{arriveToS1}}$ and $E_{\text{arriveToS2}}$ is triggering different events depending on the relative time of occurrence between them:
event $E_{S2oS3}$ is $E_{arriveToS1}; E_{arriveToS2}[[tocc(E_{arriveToS1}) - tocc(E_{arriveToS2}) < 4]]$

event $E_{S2oS4}$ is $E_{arriveToS1}; E_{arriveToS2}[[tocc(E_{arriveToS1}) - tocc(E_{arriveToS2}) >= 4]]$

The event occurrences of type $E_{S2oS3}$ and $E_{S2oS4}$ are then triggering the rules Rule S2toS3 and Rule S2toS4 respectively. Another solution would be to use the clock values in the condition part of the rules, if such clock variables are supported by the rule base. The notation used for relative event occurrences are further described in Mellin (1998).

5.1.5 Synchronisation

In Uppaal it is possible to model synchronisation between processes. Figure 5.5 shows a synchronisation between two different automata. The first automaton contains the states S1, S2 and S3 and the second contains the states P1, P2 and P3. These automata is synchronising on the channel $sync$, meaning that they may only take their transitions P2 to P3, and S2 to S3 simultaneously, not one by one.

To transform the synchronizing behaviour to ECA rules, the arrival to state S2 and P2 are derived to events of type $E_{arriveToS2}$ and $E_{arriveToP2}$. To take the synchronising transition, both an event of type $E_{arriveToS2}$ and $E_{arriveToP2}$ must have occurred, meaning
CHAPTER 5. VERIFY TRANSFORMATIONS

![Diagram of two timed automata synchronising on channel sync](image)

Figure 5.5: Two timed automata synchronising on channel sync, the action parts are executed in parallel.

that a composite event of type $E_{arriveToS2} \triangle E_{arriveToP2}$ must have occurred.

Since a synchronisation is dependent on that two or more processes are in a certain state, more than one event must have occurred in the rule base before the action transformed from the synchronizing transition can be executed. This implies that the synchronization is triggered by a composite event occurrence, and hence, the consumption policy of the target ADBMS must be considered.

It is assumed that the composite event occurrence will be consumed in chronicle context, no other automata are synchronizing on channel sync, and the event occurrences of type $E_{arriveToS2}$ and $E_{arriveToP2}$ are only raised as the system arrives to state S1 and S2. The automaton in Figure 5.5 are transformed to the following rule where $\|$ denotes that the action parts are executed in parallel:

\[
\text{ON } (E_{arriveToS2} \triangle E_{arriveToP2})
\]
\[
\text{DO } (actionS2toS3() \| actionP2toP3())
\]

If the target event monitor uses recent context, then it is not possible to derive a conjunction from the specification in Figure 5.5. The reason is that in recent context the event occurrences remain in the event history and as one occurrence of a conjunction has
CHAPTER 5. VERIFY TRANSFORMATIONS

occurred in recent context, only one of the terminator events has to occur to raise the
composite events next time. In recent context it must be explicitly expressed that the
occurrence of $E_{\text{arriveToS2}}$ and $E_{\text{arriveToP2}}$ must not have any occurrence of $E_{\text{arriveToS3}}$ or
$E_{\text{arriveToP3}}$ in-between for the composite event to be raised. In recent context the following
rule will express the behaviour of the automaton in Figure 5.5:

\[
\text{ON } (N(E_{\text{arriveToS2}}, E_{\text{arriveToS3}}, E_{\text{arriveToP2}}) \lor N(E_{\text{arriveToP2}}, E_{\text{arriveToP3}}, E_{\text{arriveToS2}}))
\]
\[
\text{DO } (\text{actionS2toS3()||actionP2toP3()})
\]

The event part of the rule is a disjunction of two non-occurrence events. If state S2 is
reached before state P2, an event occurrence of type $N(E_{\text{arriveToS2}}, E_{\text{arriveToS3}}, E_{\text{arriveToP2}})$
is initiated. The transition between S2 and S3 can not be taken until the other automaton
reaches state P2. In the rule base, it means that events of type $E_{\text{arriveToS2}}$ and $E_{\text{arriveToP2}}$
have occurred without any occurrence of type $E_{\text{arriveToS3}}$ or $E_{\text{arriveToP3}}$ in-between. The
rule that takes the system from state P2, S2 to P3, S3 is triggered and the action part is ex-
ecuted. The terminator occurrence of the triggering rule is of type $E_{\text{arriveToP2}}$, and this oc-
currence will also initiate a composite event of type $N(E_{\text{arriveToP2}}, E_{\text{arriveToP3}}, E_{\text{arriveToS2}})$,
however, this event is terminated as the system reaches state P3, meaning that the oc-
currence of type $N(E_{\text{arriveToP2}}, E_{\text{arriveToP3}}, E_{\text{arriveToS2}})$ is used, and as an event of type
$E_{\text{arriveToS2}}$ occurs next time, the composite event of type $N(E_{\text{arriveToP2}}, E_{\text{arriveToP3}}, E_{\text{arriveToS2}})$
will not terminate unless there is a new occurrence of type $E_{\text{arriveToP2}}$.

5.1.6 Limited transformation of guards

In the following sections, constructs of timed automata which are not possible to directly
transform into rules using the method proposed in (Ericsson 2002) is discussed. In cases
where alternative solutions for problematic transformations are found, alternative solutions
Several valid transitions to take

If there is more than one valid transition to take in a state, the automaton is said to be non-deterministic, since the outcome of a triggered rule may be different even if an equal set of inputs are given. If a non-deterministic timed automaton specification is derived into rules, the rule set will also become non-deterministic and if this is not a desirable result, several valid transitions in a state must be avoided.

No valid transition to take

Figure 5.6 represents a common construct in timed automata. As the model reaches state S1, the guard is evaluated. If the guard is evaluated to true \( x == 100 \) in the example, the transition between S1 and S2 is taken as soon as possible. If the guard is false, the automaton remains in state S1 until the guard becomes true, and as the guard becomes true, the transition is taken.

As the specification in Figure 5.6 is transformed into ECA rules, an important divergence between the behaviour of a timed automaton and ECA rules must be considered. As the transformed event of type \( E_{arriveToS1} \) occurs in the rule base, the condition to the associated rule is evaluated once, and if it evaluates to false, no action is executed. The
CHAPTER 5. VERIFY TRANSFORMATIONS

condition is not evaluated again until an event of type $E_{\text{arr} \text{ivTo} S_1}$ occurs next time. This means that it is not possible to transform the specification in Figure 5.6 to a rule triggered by the primitive event $E_{\text{arr} \text{ivTo} S_1}$ and with the condition $x == 100$.

This behavioural divergence appears every time a state is reached which does not contain exactly one valid transition for all cases. In the example above, a valid transition is missing for the case where the guard is evaluated to false.

Alternative solution for guards

In the work of Ericsson (2002) as well as in the work of Berndtsson et al. (1997), the problem of no valid transitions, caused by guards not covering all cases, is solved by transforming the guards into composite events. The basic idea is to replace the guard with the event occurrence which is setting the guard to true. In the previous studies, there is knowledge of the specific applications that are modelled, however, in the following solution, no knowledge of the application semantics is assumed. This makes the task even more complex, since a guard in a timed automaton for example may represent the state of a sensor (e.g. if the sensor is high or low), or an integer counter which is true for different values in different states of the automaton. The guard may also change value in different ways, for example during the guarded transition, or in another part of the system executing in parallel.

In the following, suggested solutions for modelling the behaviour of guards as composite events are presented. The guarded transitions are first reconstructed to synchronizing timed automata, and then transformed to composite events. The reason for reconstructing the automaton before the transition is performed is that it is possible to automatically verify that the reconstructed behaviour corresponds to the stated requirements. It is also easier to verify that the behaviour of the resulting rule set corresponds to the behaviour specified in the reconstructed automaton. It is assumed that it is possible to model the occurrence that changes the value of the guard (e.g. to model the event $E_{\text{sensor becomes high}}$.
instead of using the guard $sensor == \text{high}$) and that the transition that changes the guards value can synchronize with the automaton modelling the behaviour of the guard.

Guard which only changes value in the guarded transition

A simplification of the conjunction pattern described in chapter 5.2 can be used to model the guard in Figure 5.6 for transformation to the chronicle context, provided that the following preconditions are fulfilled:

I. As the guard has become true, it remains true until the guarded transition is taken.

II. The guarded transition is the only transition which changes the guard to false.

The suggested solution for chronicle context is to model the guard in a separate automaton which is synchronizing with the original automaton as shown in Figure 5.7. The upper automaton in Figure 5.7 represents the reconstructed version of the automaton in Figure 5.6 and the lower automaton represents the reconstructed guard. The reconstructed guard is synchronizing with some other automaton which is changing the guard from false to true using the channel $GuardTrue$. Given the assumptions above, the guard will only be changed to false as the transition $S1\rightarrow S2$ is taken.

The result of transforming the reconstructed guard to ECA rules in chronicle context is the following:

Rule $S1\rightarrow S2$

ON $F_{arriveToS1} \land F_{guardBecomeTrue}$

DO $ActionS1ToS2() || G3acceptToG0\rightarrow start()$

If the target execution model is executing the rules in recent context, the guard in
CHAPTER 5. VERIFY TRANSFORMATIONS

Figure 5.7: Alternative solution for simple guard that can be transformed to a conjunction in chronicle context

Figure 5.6 will be reconstructed to a disjunction of two non-occurrence events. The reconstructed timed automaton will then be transformed into the following event type:

\[
N(E_{\text{arriveToS1}}, E_{\text{arriveToS2}}, E_{\text{guardBecomeTrue}}) \lor \\
N(E_{\text{guard BecomeTrue}}, E_{\text{arriveToS2}}, E_{\text{arriveToS1}})
\]

Guard which may change value anytime

If the true guard can be falsified by some other process, or by an external occurrence, the construct shown in Figure 5.8 may be used to model the guard in Figure 5.6 instead. In this example, x is an integer value that is used as a guard for different transitions, and the guard is true for different values (\(x = 100\) or \(x = 1\)) in different transitions in the system. As soon as some transition is changing the value of x, it is required to synchronize on channel changeX. Synchronizing on this channel causes the guard automaton to evaluate the value of x, if the value is 100, an urgent transition to state \(S_{X100}\) is taken, and any automaton which is guarded by \(x = 100\) can synchronize with this transition. As the synchronizing transition is taken, the guard automaton is in state \(S_{Evaluate}\), if the
CHAPTER 5. VERIFY TRANSFORMATIONS

guard is still 100, the transition between $S_{Evaluate}$ and $S_{Xis100}$ is taken again, but if the value does not correspond to any of the guarded values, the automaton remains in state $S_{Evaluate}$. Assuming that the events transformed from the automaton in Figure 5.8 will be executed in chronicle consumption mode, the following rules will implement the specified behaviour:

\[
\text{ON } E_{\text{arriveToEvaluate}} \\
\text{IF } x == 1 \\
\text{DO } \text{actionSEvaluateToSXis1()}
\]

\[
\text{ON } E_{\text{arriveToEvaluate}} \\
\text{IF } x == 100 \\
\text{DO } \text{actionSEvaluateToSXis100()}
\]

\[
\text{ON } N(\text{E_{arriveToS1}, E_{arriveToS2}, E_{arriveToSXis1}}) \lor \\
N(\text{E_{arriveToSXis1}, E_{arriveToSEvaluate}, E_{arriveToS1}}) \\
\text{DO } \text{actionS1toS2() || actionSXis1toSEvaluate()}
\]

The non occurrence event is necessary since the $E_{\text{arriveToSXis1}}$ will remain in the event history until it is invalidated or consumed. Using a non occurrence event, the semantic that something is making the event occurrence untrue is captured, the fact that the guard became true is erased by an occurrence of type $E_{\text{arriveToSEvaluate}}$.

The solution in Figure 5.8 is using guards in the automaton simulating the original guard. The reason why guards are possible to use in this automaton is that as soon as some other automaton is changing the value of the guard $x$, or takes a transition guarded by $x$, an event occurrence of type $E_{\text{arriveToSEvaluate}}$ is re-raised which forces the rule set to
re-evaluate its conditions.

The examples above are some suggestions for how to reconstruct the automaton if the semantics of guards have to be used. Transforming guarded transition to rules becomes complex in the general case, since the result of the transformation is depending on several parameters which may not be known if the guarded transition is transformed by a machine. It is also dependent on that the semantic of the guard can be rewritten to the occurrence when the guard became true. In for example chronicle context, this solution may be awkward, since the event occurrence stating that the guard has become true is consumed as it is used to trigger an event occurrence. If the guard is not changed to false by the action triggered by the rule, the occurrence that the guard is true must be re-raised and this causes that the event history in the rule base is not consistent with the actual history of event occurrences.

**Outline of algorithm for reconstructing guards**

To be able to use guards in models that should be transformed into ECA rules using the method proposed in Ericsson (2002), an algorithm is needed that automatically transform
guards in an arbitrary specification to rules. In a final solution, the guards may be transformed without the intermediate step of reconstructing the guard automaton, however, in a first attempt, reconstructing the guards is a necessary step to verify the correctness of the resulting rule set. The algorithm for reconstruction must search the model for all guards. For each guard it must have the ability to decide if several transitions are using the same variable as guard and in which transitions the guard changes its value. Given this information, the timed automaton model can be reconstructed. An outline of the algorithm for automatically reconstructing guards is presented in pseudo code:

Find the set $X$ of all variables that are serving as guards

for each guard $x$ in $X$ do begin
    Find the set $Y$ of all transitions that are changing the value of $x$
    Find the set $Z$ of all transitions that are guarded by $x$
    If ($x$ is only used as a guard for one value and $Z == Y$) then begin
        Construct a guard automaton according to Figure 5.7 (for chronicle context)
        for all $Y$ do
            Synchronize with guard automaton
    end
    If ($x$ is used as a guard for several different values) then begin
        Construct a guard automaton according to Figure 5.8 (for chronicle context)
        with one synchronisation for each guarded value of $x$
        for all $Y$ do
            Synchronize with guard automaton (on channel ChangeX!) 
        for all $Z$ do
            Synchronize with guard automaton
    end
CHAPTER 5. VERIFY TRANSFORMATIONS

end

5.2 Specifying patterns in timed automata

In the following section, patterns for specifying composite events in timed automata will be identified. Occurring events are assumed to be sporadic, which means that they have a minimum interarrival time, however, the exact time of occurrence and the orders in which the events will occur are not known a priori.

The identified patterns will facilitate transformation of composite events from an arbitrary timed automaton, since they may be automatically identified in a larger system. The identified patterns are also useful when specifying an event triggered system in timed automata, where for example the order of a set of event occurrences is not known.

Each composite event pattern is specified in the contexts chronicle and recent. As a first step, the pattern is presented without taking the order of event occurrences of the same type into consideration. This is sufficient for recent context where used event occurrences are not consumed. As a second step, the event occurrences in chronicle context are assumed to carry important event parameters, implying that the importance of considering the order in which the event occurs increases. As a third step, the acceptance language of the pattern will be specified and as a fourth step, a timed automaton pattern for expressing operators in chronicle context with expiration times is modelled.

5.2.1 Conjunction

As an event of type $E_1$ and $E_2$ occurs within a specified time period, a composite event of type $E = E_1 \triangle E_2$ is raised.

Let $G$ be a model such that
\[ \mathcal{G} \models \text{occ}(E_1 \triangle E_2, [t, t']) \text{ iff } \exists [t_1, t_2]((\text{occ}(E_1, [t, t_1]) \land \text{occ}(E_2, [t_2, t'])) \lor (\text{occ}(E_2, [t, t_1]) \land \text{occ}(E_1, [t_2, t'])) \land t \leq t_1 \land t_2 \leq t' \lor (\text{occ}(E_2, [t, t']) \land \text{occ}(E_1, [t_1, t_2]) \land t \leq t_1 \land t_2 \leq t') \]

In other words, there is an occurrence of both event type \( E_1 \) and \( E_2 \) within the time period that starts with \( t \) and ends with \( t' \).

### Conjunction in chronicle context

Figure 5.9 shows an automaton which signals an event occurrence of type \( E_{ab} \) as an event of type \( E_a \) and \( E_b \) have occurred in chronicle context. The time interval is not considered for the composite event in Figure 5.9, so the event occurrences are signalled independently of time expirations. The automaton in Figure 5.9 is not consuming event occurrences of the same type in chronicle order either. This is solved in the automaton shown in Figure 5.10 where arrays are used to represent event parameters used in chronicle order.

In Figure 5.9 the counters \( ca \) and \( cb \) are used to count the number of event occurrences of type \( E_a \) respectively \( E_b \). As the automaton is in state \( S_1 \) and an event of type \( E_a \) occurs, the automaton models this occurrence by synchronizing with an automaton raising the event of type \( E_a \). (The channel \( E_a \) must synchronize with channel \( E_a \) in another automaton, that is raising the event occurrence. Automatons synchronizing on a channel must take their transitions simultaneously.) As \( \{ \Gamma(E_a, [t, t_1]), \Gamma(E_b, [t_2, t']) \} \), or \( \{ \Gamma(E_b, [t, t_1]), \Gamma(E_a, [t_2, t']) \} \) has occurred where \( t_1 \leq t_2 \), the automaton is raising an event of type \( E_{ab} \) that will synchronize with some automaton waiting for the event of type \( E_{ab} \) to occur. In the following chapters, the fact that the pattern is synchronising with another automaton that is raising the event, is implicit as an event occurs.

As the automaton for detecting occurrences of type \( E_a \) and \( E_b \) reaches state \( S3 \), there is at least one occurrence of type \( E_a \) and at least one occurrence of type \( E_b \) that have not
CHAPTER 5. VERIFY TRANSFORMATIONS

![Diagram of a automaton](image)

Figure 5.9: Composite event pattern of type $E_{ab} = E_a \triangle E_b$ in chronicle context where the occurrence order between events of the same types are not considered.

participated in a composite event occurrence of type $E_{ab}$. If there are exactly one unused occurrence of type $E_a$ and exactly one unused occurrence of type $E_b$ in the event history, the automaton is taking the transition back to S0. If there are more than one occurrence of type $E_a$ (or $E_b$), the automaton is taking the transition back to S1 (or S2).

**Considering order of occurrence**

To verify that the automaton pattern for conjunction can be useful for chronicle context, parameters represented as integer values are attached to each event occurrence. The parameter values are stored in an array in the order of the occurrence of events of the same type, and the parameters attached to the composite event occurrence are formed from the parameters of the constituent event occurrences. A timed automata solution for this is shown in Figure 5.10.

For each event occurrence of type $E_a$, (or $E_b$), the counters $ca$, (or $cb$) are increased. The value of the counters is assigned to the event occurrences as parameters. As there is

68
an occurrence of both type $E_a$ and type $E_b$, a composite event of type $E_{ab}$ is raised, and is
given the sum of the parameters of its constituent events as its own parameter value. The
counters $a$ and $b$ are increased as a composite event occurs, counting the number of used
event occurrences of type $E_a$ and $E_b$. Since this is a model where no event occurrences are
invalidated, the number of composite event occurrences are equal to $a$ and $b$ as the system
is in state S3.

Acceptance language for conjunction in chronicle context

The acceptance language for the automaton pattern simulating conjunction in chronicle
context can not be expressed as a regular expression. The automaton must keep track on
how many instances of a particular event type that has occurred, and this is not possible
to express in a regular expression. However, the first time the automaton reaches the ac-
ceptance state can be expressed as a regular expression.
$$
\mathcal{L}(\mathcal{M}) = ((E_a)^+, E_b \cup (E_b)^+ E_a)
$$

This is an expression of how to reach the acceptance state for one composite event occurrence. In the first case, one or more event occurrences of type $E_a$ must occur, followed by one occurrence of type $E_b$. In the second case, one or more occurrence of an event of type $E_b$ must occur followed by one event occurrence of type $E_a$. The union between them states that one of the first and second cases is enough to reach the acceptance state. However, how to reach the acceptance state next time and raise a new composite event is depending on the number of previous event occurrences and consumption of event occurrences, and this is not possible to express using the regular grammar. The regular expression for the first conjunction in chronicle context is correct if each initiator event is starting a new process for its detection.

**Conjunction with expiration times in chronicle context**

Adding expiration times to the composite events means that the composite event has a deadline. If the duration between the occurrences of the initiator event and the terminating event exceeds a time-limit, the event composition is interrupted.

Adding expiration times to conjunction in chronicle context makes the pattern for detecting composite event occurrences more complex. The timed automaton must save time stamps for each event occurrence and use these time stamps for later evaluations. A timed automaton model for this behaviour is shown in Figure 5.11.

In the pattern shown in Figure 5.11, there is a variable named $tNow$ representing the current global time. The counters $a$ and $b$ keeps the index of the oldest unused and un-invalidated event occurrence of type $E_a$ and $E_b$. The time stamp for the initiator event considered is stored in $tStamp$. In the example shown in Figure 5.11, the expiration time is 10 time units. As for example an event of type $E_b$ occurs, the global time of the
Figure 5.11: Composite event of type $E_{ab} = E_a \triangle E_b$ in chronicle context with expiration time
event occurrence is saved in the array Bpar at position \( d \) as a time stamp parameter. If there are more event occurrences of type \( E_b \) before next event occurrence of type \( E_a \), their timestamps are also stored. If there are no occurrence of event type \( E_a \) within the specified time period (10 time units), the first unused occurrence of event type \( E_b \) is invalidated.

As an initiator is invalidated, there are two choices, in the first case, there are more unused and un-invalidated occurrences of type \( E_b \) and in the second case, this was the only occurrence of type \( E_b \) that was not used or invalidated. In the first case, the automaton is staying in state S2 and the timestamp for the next unused and un-invalidated occurrence of type \( E_b \) is used to check for the next expiration time and in the second case, the automaton is taking the transition back to its starting state S0.

**Conjunction in recent context**

In the recent context, only the most recent event occurrences are considered. The timed automata pattern for the conjunction operator in recent context is shown in Figure 5.12. The conjunction operator in recent context is less complex than the chronicle since there is no need to save all unused event occurrences. As soon as there is a new occurrence of an event of type \( E_a \) or \( E_b \), the old instance of that event type is deleted and replaced with the new instance, and a new instance of the composite event of type \( E_{ab} \) is raised.

**Acceptance language for conjunction in recent context**

The acceptance language for the automaton pattern modelling conjunction in recent context with no time constraints can be expressed as a regular expression.

The automaton is accepting any permutation of event occurrences of type \( E_a \) and \( E_b \) where there is at least one instance of type \( E_a \) and at least one instance of type \( E_b \).

\[
\mathcal{L}(M) = ((E_a)^+, E_b \cup (E_b)^+ E_a)(E_a \cup E_b)^*
\]
This means that the automaton pattern for a conjunction of \( E_a \triangle E_b \) in recent context is first accepting at least one occurrence of type \( E_a \) (or \( E_b \)) followed by a single occurrence of type \( E_b \) (or \( E_a \)). After that, the automaton is accepting any permutation of occurrences of event type \( E_a \) and \( E_b \).

### 5.2.2 Disjunction

As an even event of type \( E_1 \) or \( E_2 \) has occurred within a specified time period, a composite event of type \( E_1 \uparrow E_2 \) has occurred.

Let \( \mathcal{G} \) be a model such that

\[
\mathcal{G} \models \text{occ}(E_1 \uparrow E_2, [t, t']) \iff \text{occ}(E_1, [t, t']) \vee \text{occ}(E_2, [t, t'])
\]
Figure 5.13: Composite event of type $E_a \lor E_b$ in chronicle and recent context.

In other words, there is an occurrence of an event of type $E_1$, or $E_2$ within the time period that starts with time $t$ and ends with time $t'$.

**Disjunction in chronicle and recent context**

The pattern for a composite event with a disjunction operator is less complex than the pattern for a conjunction operator. The composite event of type $E_a \lor E_b$ shall be raised if either an event of type $E_a$, or $E_b$ occurs. As shown in Figure 5.13 there is no need for saving old occurrences of events even in the chronicle context since there will be no unused events in the event history. As soon as there is an occurrence of one of the constituent event types, the pattern moves from state $S_0$ to state $S_1$, and as soon as it reaches state $S_0$, a composite event is formed and the pattern raises an event of type $E_{ab} = E_a \lor E_b$ as it takes the transition back to $S_0$. This means that the pattern for disjunction is equal in both recent and chronicle context, given that there are no events occurring simultaneously.

**Acceptance language for chronicle and recent context**

The acceptance language for chronicle and recent context is any permutation of event occurrences of type $E_a$ and $E_b$. 

74
\[ \mathcal{L}(\mathcal{M}) = (E_a \cup E_b) ^+ \]

The automaton specifying the behaviour of a disjunction operator is accepting any permutation of \( E_a \) and \( E_b \) since it raises a new composite event as soon as an event occurrence of type \( E_a \), or \( E_b \) is signalled.

### 5.2.3 Sequence

As an event of type \( E_1 \) occurs before an event of type \( E_2 \), an event occurrence of type \( E_1; E_2 \) may be raised.

Let \( \mathcal{G} \) be a model such that

\[
\mathcal{G} \models \text{occ}(E_1; E_2, [t, t']) \iff \\
\exists t_1 \leq t_2 (\text{occ}(E_1, [t, t_1]) \land \text{occ}(E_2, [t_2, t'])) \land (t_1 - t > 0 \lor t' - t_2 > 0)
\]

In other words, there is an occurrence of an event of type \( E_1 \) followed by an event occurrence of type \( E_2 \) within the time period that starts with \( t \) and ends with \( t' \).

**Sequence in chronicle context**

The automaton in Figure 5.14 shows the behaviour of a sequence operator of type \( E_{ab} = E_a; E_b \) in chronicle context. The start state of the automaton in Figure 5.14 is \( S0 \). As the system is in state \( S0 \), there are no unused or un-invalidated occurrences of type \( E_a \), or \( E_b \) in the event history.

The only type of event occurrences that can initiate the composite event is \( E_a \) and the only type that can terminate the composite event is \( E_b \). This means that if an event of type \( E_b \) occurs in state \( S0 \), it is invalidated since it can not be used to terminate any occurrence
Figure 5.14: Composite event of type $E_{ab} = E_a; E_b$ in chronicle context.

As an event occurrence of type $E_a$ occurs in state S0, the pattern takes the transition to state S1.

In state S1, there is at least one event occurrence of type $E_a$ in the event history that are not invalidated or used. This means that it is possible to accept the event occurrence that will terminate the composite event, but it is also possible to accept a new initiator for a new occurrence of the composite event of type $E_{ab}$. If a composite event of type $E_a$ occurs, the pattern remains in state S1 and the parameters for the new initiator occurrence are saved for later use. As there is an occurrence of type $E_b$, the transition is taken to state S2, where the composite event of type $E_{ab}$ is raised.

If there exists more than one unused occurrence of type $E_a$ in the event history in state S2, the transition from S2 to S1 is taken, since there is already an initiator occurrence for the composite event and a new occurrence of type $E_b$ is needed. If there is exactly one occurrence of type $E_a$ and exactly one occurrence of type $E_b$ that are not used, or invalidated in the event history, the transition between S2 and S0 is taken as the composite event of type $E_{ab}$ is raised.
CHAPTER 5. VERIFY TRANSFORMATIONS

Acceptance language for sequence in chronicle context

As for conjunction in chronicle context, the acceptance language for sequences in chronicle context can not be expressed in a regular language, since there is a need to count the number of occurrences of type $E_a$. However, the acceptance language for identifying one sequence occurrence is possible to express in a regular language:

$$L(M) = (E_b)^*, (E_a)^+, E_b$$

This is an expression for how to reach one composite event of type $E_{ab} = E_a; E_b$. The first occurrences of type $E_b$ are unimportant so there may exist zero or an arbitrary number, however, there must be at least one occurrence of type $E_a$, followed by an occurrence of type $E_b$ for an event of type $E_{ab}$ to be raised.

Sequence with expiration times in chronicle context

If there is an occurrence of the initiator type, and an occurrence of the terminator type does not arrive within a specified time period, the composite event may be useless for the system and the initiator occurrence should be invalidated. A timed automaton pattern for a sequence with expiration times in chronicle context is shown in Figure 5.15.

In state S0, event occurrences of type $E_b$ are invalidated, since they are only used as terminators in this pattern. However, the counters must be increased as the events occur to be able to identify the earliest unused event occurrences that are also not invalidated. As an occurrence of type $E_a$ is raised, the pattern takes the transition between S0 and S1. The counters are increased and the current time is saved in the array Apar in a similar way as for conjunction with expiration times.
Figure 5.15: Composite event of type \( E_{ab} = E_a; E_b \) in chronicle context with expiration times.

**Sequence in recent context**

The timed automaton pattern for a sequence in recent context is shown in Figure 5.16. State S0 is the start state of the pattern, and since an event occurrence of type \( E_a \) must initiate the composite event, occurrences of type \( E_b \) are invalidated until the first event of type \( E_a \) occurs. As an event of type \( E_a \) occurs, the transition to S1 is taken. In this state, the parameters of a prior occurrence of type \( E_a \) will be overwritten by the parameters of the new occurrences of type \( E_a \), since it is only the most recent occurrence that are of interest here. As an event of type \( E_b \) occurs in state S1, the transition to S2 is taken and a composite event is immediately formed as the pattern takes the transition to state S3. In state S3, a new composite event is raised each time a terminator event occurs. The composite event will use the most recent occurrence of type \( E_a \) and the terminator of type \( E_b \) to form the composite event.
CHAPTER 5. VERIFY TRANSFORMATIONS

![Composite event diagram]

Figure 5.16: Composite event of type $E_a\cdot E_b$ in recent context.

Acceptance language for sequence in recent context

The acceptance language for a sequence pattern in recent context can be formed as a regular expression.

$$L(M) = (E_b)^*, E_a, ((E_a)^*, (E_b))^+$$

For a start there may be a finite number of event occurrences of type $E_b$. After that, one or more occurrences of type $E_a$ must occur, followed by one or more permutations of event occurrences where zero or more event occurrences of type $E_a$ is followed by one event occurrence of type $E_b$.

5.3 Equivalence of semantics

To prove the equivalence of semantics between timed automaton patterns and their corresponding composite event expressions, a formalized schema for event composition will be
used (Mellin & Andler 2002). (The notation used is described in more detail in section 2.4.3.) The event composition schema describes rules for event compositions for different operators in different contexts. State changes of the rule set are defined as occurred events together with the set of events that are used, or invalidated for all the event composition operators. This means that as an event occurs, becomes used, or becomes invalidated, the system is transformed to a new state.

The aim of this section is to express each state of timed automata patterns in sets of occurred, used and invalidated events, to verify that each event occurrence causes an equivalent change of state in both the timed automaton specification and the rule set. The proofs of equivalence between timed automaton pattern and its corresponding composite event will be performed for conjunction, disjunction and sequence in chronicle context. The formal definitions of the chronicle context predicate and rule predicates in the following sections are reprinted from Mellin (2003).

5.3.1 Assumptions

In the following proofs it is assumed that only occurrences of event type $E_a$ and $E_b$ affects the behaviour of the operator patterns and the composite events. This is a fair assumption since $E_a$ and $E_b$ is the only events included in the alphabet of the operator patterns and they are the only event types that will affect the composite event occurrences derived from the operator pattern. As an acceptance state is reached, the composite event pattern returns to some previous state and is signalling the occurrence of the composite event $E_{ab}$.

An event is said to occur in the timed automaton pattern as the automaton synchronizes with another automaton on channel Ea? or Eb?. As a composite event has occurred, the composite event pattern raises the composite event by synchronizing on channel Eab!.

It is assumed that no event occurrence can come in between as a composite event has occurred, it is raised immediately. Otherwise in e.g. conjunction it must be possible to
CHAPTER 5. VERIFY TRANSFORMATIONS

accept different occurrences of Ea and Eb in each accepting state.

The environment is assumed to be unpredictable, i.e. the order of event occurrences is not known a priori.

5.3.2 Chronicle Context Predicate

As shown in previous sections, the set of composite events that are raised by a set of primitive (or composite) event occurrences are depending on the current context (consumption policy). In the following sections, formal definitions for operator rules will be explained and used for proving the correctness of the transformed behaviour between timed automaton patterns and composite events.

One of the predicates that must be true in each operator rule is the context predicate. The context predicate is needed to select the initiator and terminator occurrence with respect to used and invalidated event occurrences. In chronicle context it is the earliest event occurrence that is not used nor invalidated, that will be used as initiator and terminator to form the composite events.

Since different consumption policies can choose different constituent event occurrences for the same operator, the context predicate for different consumption policies are different. In this subsection the chronicle context predicate will be explained, since it will be used in proofs in the following subsections. The operator rules and context predicate for chronicle context as well as its constituent predicates are rewritten from Mellin (2003).

Recall from section 2.4.3 that \( E_\alpha \) is the set of event types that can initiate the composite event occurrence of type E. \( G \) is the set of events that have occurred so far, \( \gamma \) is an event occurrence and \( exp(\gamma) \) states that the expiration time of the initiated rule has expired. In the following proofs \( X \xrightarrow{\lambda} Y \) stands for \( X \) is a lexical substitution for \( Y \). To improve readability of the rules, the lexical substitution \( \mathcal{X} \cup \mathcal{Y} \xrightarrow{\lambda} \mathcal{X}' = \mathcal{X} \cup \mathcal{Y} \) is used.
CHAPTER 5. VERIFY TRANSFORMATIONS

Let \( \mathcal{ALL}_\alpha \overset{\lambda}{=} \{ \gamma \mid \gamma \in \mathcal{G} \land (\text{type}(\gamma) \in \mathcal{E}_\alpha) \land \neg \text{exp}(\gamma) \} \)

In other words, \( \mathcal{ALL}_\alpha \) is the set of all event occurrences which are of an event type that can initiate the composite event of type E. It is also the case that the expiration time for the event possible to compose by the initiator has not expired.

Let \( \mathcal{ALL}_\omega \overset{\lambda}{=} \{ \gamma \mid \gamma \in \mathcal{G} \land (\text{type}(\gamma) \in \mathcal{E}_\omega) \land \neg \text{exp}(\gamma) \} \)

In other words, \( \mathcal{ALL}_\omega \) is the set of occurred events which are of the event type that can terminate the composite event of type E. It is also the case that the expiration time for the event that is possible to compose by the terminator has not expired.

Let \( \mathcal{USED}_\alpha \overset{\lambda}{=} \text{iot}(\mathcal{U}_\alpha(E)) \)

Let \( \mathcal{USED}_\omega \overset{\lambda}{=} \text{iot}(\mathcal{U}_\omega(E)) \)

In other words, \( \mathcal{USED}_\alpha \) is the set of occurred events of the event type that can initiate the composite event of type E and that are already used to compose another instance of the composite event of type E. \( \mathcal{USED}_\omega \) is the set of event occurrences that is already used to terminate another instant of the composite event of type E. (Recall that \( \mathcal{U}_\alpha \) is the set \( \langle \gamma_\alpha, \gamma \rangle \) where \( \gamma_\alpha \) is the initiator of \( \gamma \), \( \mathcal{USED}_\alpha \) does only contain event occurrences of the initiator type, in this case \( \gamma_\alpha \), and vice versa for \( \mathcal{USED}_\omega \))

Let \( \mathcal{CONS} \overset{\lambda}{=} \text{iot}(\mathcal{U}_\alpha(E)) \cup \mathcal{I}_\alpha(E) \cup \text{iot}(\mathcal{U}_\omega(E)) \cup \mathcal{I}_\omega(E) \)

In other words, \( \mathcal{CONS} \) is the set of event occurrences that are of an event type that
CHAPTER 5. VERIFY TRANSFORMATIONS

can terminate, or initiate the composite event E, but the occurrences are already used or invalidated.

\[
\text{\textsc{available}}_\alpha \triangleq \text{\textsc{all}}_\alpha \setminus \text{\textsc{cons}}
\]

\text{\textsc{available}}_\alpha is the set of the earliest available occurrences of the initiators for the composite event of type E that are not invalidated or used.

\[
\text{\textsc{available}}_\omega \triangleq \text{\textsc{all}}_\omega \setminus \text{\textsc{cons}}
\]

\text{\textsc{available}}_\omega is the set of the earliest available potential terminator occurrences for the composite event of type E that are not invalidated or used.

The predicates above are used in the following formula describing the context predicate for chronicle context

\[
\text{\textit{chronicle}}_\alpha (E, \gamma_\alpha, \mathcal{E}_\alpha) \triangleq \forall \gamma \in \text{\textsc{available}}_\alpha (\gamma_\alpha \neq \gamma \Rightarrow \gamma \triangleright \gamma_\alpha) \land \gamma_\alpha \in \text{\textsc{available}}_\alpha
\]

\[
\text{\textit{chronicle}}_\omega (E, \gamma_\omega, \mathcal{E}_\omega) \triangleq \forall \gamma \in \text{\textsc{available}}_\omega (\gamma_\omega \neq \gamma \Rightarrow \gamma \triangleright \gamma_\omega) \land \gamma_\omega \in \text{\textsc{available}}_\omega
\]

In other words, the initiator occurrence should be the earliest available initiator occurrence that are not already used, invalidated or expired. The terminator occurrence should be the earliest available occurrence of the type that can terminate the composite event and that are not already used, invalidated, or expired.

The E is the type of the composite event, \(\mathcal{E}_\alpha\) is the set of event types that can initiate E, \(\mathcal{E}_\omega\) is the set of event types that can terminate E, \(\gamma_\alpha\) is the current potential initiator
CHAPTER 5. VERIFY TRANSFORMATIONS

occurrence and $\gamma_\omega$ is the current potential terminator occurrence.

The context predicate used in the operator rules for chronicle context is a conjunction of the initiator and terminator consumption predicates for chronicle context. Let $X$ be the predicate then:

$$\mathcal{X}(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega) \triangleq chronicle_\alpha(E, \gamma_\alpha, \mathcal{E}_\alpha) \land chronicle_\omega(E, \gamma_\omega, \mathcal{E}_\omega)$$

The context predicate $\mathcal{X}(E, \gamma_\alpha, \gamma_\omega, \mathcal{E}_\alpha, \mathcal{E}_\omega)$ is used in the following operator rules, since it is only true as the event occurrences $\gamma_\alpha$ and $\gamma_\omega$ are the earliest event occurrences available of the specified types.

5.3.3 Operator predicate

In the event schema developed by Mellin (2003), each operator type is defined by operator rules. Each operator rule consists of range predicates, context predicates and hypothesis. If these are all true, the conclusion is to update the different event histories and hence, a composite event is raised and the system is transformed to a new state.

The range predicate assures that the referenced event occurrences are of specific event types, the context predicate for chronicle context is discussed in the previous subsection and the hypothesis for the rule consists of the predicates for event semantics defined in section 2.2.1.

The verification of equivalence between the behaviour of the timed automaton pattern and the composite event is performed in the following steps:

- Describe the operator predicate of the composite event in text and formally.
- Formalize important variables of the timed automaton pattern.
Figure 5.17: Composite event of type $E_{ab} = E_a \triangle E_b$ in chronicle context considering order

- Formalize the invariant of each timed automaton state.
- Show that all possible inputs in each state correspond to the invariant and show that each signalling of a composite event in the automaton corresponds to a signalling of a composite event in the rule base.

### 5.3.4 Verifying conjunction pattern

In this subsection the equivalence of the behaviour between event occurrences composed by a conjunction operator and the timed automaton conjunction pattern presented in section 5.2.1 will be verified. To make it easier for the reader, the conjunction pattern is reprinted in Figure 5.17 in this section. Firstly the formalized operator rule will be explained.
Present the operator predicate in text and formally.

The operator rule for conjunction is defined by Mellin (2003) to be:

Let $\gamma = \Gamma(E, [\text{start}(\text{span}(\gamma_1)), \text{end}(\text{span}(\gamma_2))])$ in the conjunction rule c[l]:

$$\forall E_1 \triangle E_2 \in \mathcal{E}_m \exists \gamma_1, \gamma_2 \in \mathcal{G} : \text{type}(\gamma_1) = E_1 \land \text{type}(\gamma_2) = E_2 \land$$

$$\gamma_1 \prec \gamma_2 \land \mathcal{X}(E_1 \triangle E_2, \gamma_1, \gamma_2, \{E_1\}, \{E_2\})$$

\[c[l]\]

$\mathcal{G} \cup = \{\gamma\} \quad \mathcal{U}_a(E_1 \triangle E_2) \cup = \{\langle\gamma_1, \gamma\rangle\} \quad \mathcal{U}_o(E_1 \triangle E_2) \cup = \{\langle\gamma_2, \gamma\rangle\}$

The range predicate of the rule states that for all occurrences of the composite event type $E_1 \triangle E_2$ which are monitored, there is an event occurrence of type $E_1$ and $E_2$ where the occurrence of type $E_1$ precedes the occurrence of type $E_2$. The $\mathcal{X}(E_1 \triangle E_2, \gamma_1, \gamma_2, \{E_1\}, \{E_2\})$ is the chronicle context predicate described in section 5.3.2, which states that the event occurrences are the earliest available event occurrences of type $E_1$ and $E_2$. Since both the constituent event occurrences can be initiator and terminator in a composition, a complementing rule is needed which handles the case where the event occurrences are in opposite order.

86
CHAPTER 5. VERIFY TRANSFORMATIONS

Let $\gamma = \Gamma(E, [\text{start}(\text{span}(\gamma_2)), \text{end}(\text{span}(\gamma_1))])$ in the conjunction rule $c[r]$:

$$\forall E_1 \triangle E_2 \in \mathcal{E}_m \exists \gamma_1, \gamma_2 \in \mathcal{G} : type(\gamma_1) = E_1 \land type(\gamma_2) = E_2 \land$$

$$\gamma_2 \prec \gamma_1 \land \mathcal{X}(E_1 \triangle E_2, \gamma_2, \gamma_1, \{E_2\}, \{E_1\})$$

$$\mathcal{G} \cup = \{\gamma\} \quad \mathcal{U}_a(E_1 \triangle E_2) \cup = \{\langle \gamma_2, \gamma \rangle\} \quad \mathcal{U}_\omega(E_1 \triangle E_2) \cup = \{\langle \gamma_1, \gamma \rangle\}$$

If one of the rules $c[l]$ and $c[r]$ are true, a composite event of type $E_1 \triangle E_2$ is raised and the conclusion part of the rule is executed.

**Formalize important variables of the timed automaton pattern**

The automaton in Figure 5.17 contains four counters. The definitions for the counters are:

$$ca \triangleq \left| \{\gamma_a \mid \gamma_a \in \mathcal{G} \land type(\gamma_a) = E_a\} \right|$$

$$cb \triangleq \left| \{\gamma_b \mid \gamma_b \in \mathcal{G} \land type(\gamma_b) = E_b\} \right|$$

$$a \triangleq \left| \{\gamma_a \mid \gamma_a \in \mathcal{G} \land type(\gamma_a) = E_a \land (\gamma_a \in \text{iota}(\mathcal{U}_a) \lor \gamma_a \in \mathcal{I}_a)\} \right|$$

$$b \triangleq \left| \{\gamma_b \mid \gamma_b \in \mathcal{G} \land type(\gamma_b) = E_b \land (\gamma_b \in \text{iota}(\mathcal{U}_\omega) \lor \gamma_b \in \mathcal{I}_\omega)\} \right|$$

In other words, $ca$ is the number of all occurrences of event type $E_a$, $cb$ is the number of all occurrences of event type $E_b$, $a$ is the number of all event occurrences of type $E_a$ that are used or invalidated, and $b$ is the number of all occurrences of type $E_b$ that are used or invalidated.

Since the conjunction operator in chronicle context requires one new occurrence of type $E_a$ and one new occurrence of type $E_b$, and expiration times are not considered, the number of occurrences of type $E_a$ and type $E_b$ that are used as initiators or terminators are always equal ($a = b$). The pattern in Figure 5.17 does not take expiration times into
consideration so no event occurrences of type $E_a$ or $E_b$ will be invalidated. This means that the predicate $\text{CONS}$ consists of all event occurrences of type $E_a$ and $E_b$ that have been used to form a composite event and the number of elements in this set is $a + b$. The number of available occurrences of for example $E_a$ is $ca-a$ since this is the number of occurred events of type $E_a$ minus the number of used event occurrences of type $E_a$.

As a composite event is formed, the earliest available occurrence of type $E_a$ and $E_b$ should be used to form the composite event. Since events are used in chronicle order, all event occurrences that are used of one type, precedes all event occurrences that are unused of the same type. Since $a$ is the number of all used event occurrences of type $E_a$ and $b$ is the number of all used occurrences of type $E_b$, the index of the earliest unused occurrence of type $E_a$ is $a + 1$ and the index of the earliest unused occurrence of type $E_b$ is $b + 1$.

**Describe invariants of timed automata states in event algebra**

The automaton in Figure 5.17 shows a timed automaton pattern for a conjunction. For each state of the automaton in Figure 5.17 an invariant is defined based on the occurred, invalidated and used event occurrences of the event types included in the composite event.

The automaton in Figure 5.17 describes the composite event of type $E_{ab} = E_a \triangle E_b$. Since this is a conjunction, both $E_a$ and $E_b$ can be initiator, or terminator types. To increase the readability of the proof table, the notation $e^t_j$ is used to denote the occurrence of an event with index $j$ at time $t$, since this is a shorter expression for small examples of event occurrences and occupies less space in the following tables.

The set of event types that can initiate or terminate $E$ is $E_a$ and $E_b$. In other words:

$$\mathcal{E}_a \triangleq \{ E \mid E \in \mathcal{E}_m \land (E = E_a \lor E = E_b) \}$$
$$\mathcal{E}_\omega \triangleq \{ E \mid E \in \mathcal{E}_m \land (E = E_a \lor E = E_b) \}$$
CHAPTER 5. VERIFY TRANSFORMATIONS

Invariant in state $S0$:
\[
\mathcal{INV}_S \models \mathcal{AIL}_\alpha \cup \mathcal{AIL}_\omega = \emptyset
\]

This means that in state $S0$, all prior occurrences of type $E_a$ and $E_b$ are invalidated, or used. $S0$ is the start state of the conjunction pattern and the invariant is also true as no events of type $E_a$, or $E_b$ has occurred in the system.

\[
\mathcal{IS}_1 \models (\mathcal{AIL}_\alpha = \{ \gamma_a \mid \gamma_a \in \mathcal{G} \land \text{type}(\gamma_a) = E_a \}) \land (ca - a > 0) \land \\
|\mathcal{AIL}_\alpha| = ca - a
\]

Invariant in state $S1$:
\[
\mathcal{INV}_S \models \mathcal{IS}_1 \land cb - b = 0
\]

This means that in state $S1$ there are $ca-a$ event occurrences of type $E_a$ that are not invalidated or used (the number of occurred events of type $E_a$ minus the number of used events of type $E_a$) and there are no occurrences of type $E_b$ that are not invalidated or used. This means that no composite event is formed in state $S1$ according to the conjunction rule, since there are no unused and un-invalidated occurrence of type $E_b$ in the event history $\mathcal{G}$.

\[
\mathcal{IS}_2 \models (\mathcal{AIL}_\alpha = \{ \gamma_b \mid \gamma_b \in \mathcal{G} \land \text{type}(\gamma_b) = E_b \}) \land cb - b > 0 \land \\
|\mathcal{AIL}_\alpha| = cb - b
\]

Invariant in state $S2$:
\[
\mathcal{INV}_S \models \mathcal{IS}_2 \land ca - a = 0
\]

This means that in state $S2$ there are $cb-b$ number of event occurrences of type $E_b$ that
are not invalidated or used and there are no occurrences of event type \(E_a\) that are not invalidated or used.

The invariant of state S3 is a disjunction:

\[
\mathcal{I} \land S3 \triangleq (\mathcal{I} \land \mathcal{N} = \{ \gamma_b \mid \gamma_b \in G \land type(\gamma_b) = E_b \}) \land (cb - b = 1) \land |\mathcal{N}| = cb - b) \lor
\]
\[
(\mathcal{I} \land \mathcal{N} = \{ \gamma_a \mid \gamma_a \in G \land type(\gamma_a) = E_a \}) \land |\mathcal{N}| = ca - a \land (ca - a - 1))
\]

The invariant for state S3 states that there is at least one event occurrence of both type \(E_a\) and \(E_b\) that are not yet used nor invalidated. It is either the case that there is one or more initiator occurrences of type \(E_a\) and one terminator occurrence of type \(E_b\), or vice versa. The missing part from the precondition of the operator rule is the context predicate that decides which instance of the occurred event that shall be used to compose the composite event.

The invariant of state S3 together with the definition of the variables \(a\), \(b\), \(ca\) and \(cb\) is the precondition for the conjunction rule. It states that the earliest available event occurrence of type \(E_a\) \((a + 1)\), or \(E_b\) \((b + 1)\) is the initiator of the composite event and the earliest available occurrence of type \(E_a\), or \(E_b\) that is not of the same type as the initiator event, is the terminator event. If the invariant for state S3 is true, a composite event occurrence of type \(E_{ab}\) is formed and the following assignments are done in the rule base, which correspondingly makes the automaton to take a transition from S3 to S0, S1, or S2 depending on the current state of the event histories.

Conclusion assignments if the composite event is formed:

\[
G \cup = \{ \gamma \} \quad U_a(E_a \triangle E_b)\cup = \{ (\gamma_a, \gamma) \} \quad U_\omega(E_a \triangle E_b)\cup = \{ (\gamma_b, \gamma) \}
\]
CHAPTER 5. VERIFY TRANSFORMATIONS

In state S3, the composite event is raised and the earliest occurrence of event type $E_a$ and $E_b$ is added to the event histories $USED_{\alpha}$ or $USED_{\omega}$ depending on which one of them is the initiator and which one of them is the terminator occurrence.

Show that all possible inputs for each state corresponds to the state invariants

To verify that the behaviour of the automaton is equal to the behaviour of the derived rule, the state changes caused by sequences of possible event occurrences is formalized and compared. In the following tables, the changes in the event history and timed automaton, caused by possible event occurrences in each state are recorded.

For each state, a table shows the possible event occurrences and how these affects the event histories and the timed automaton states and variables. The first column in each table is named $G$ and shows the event added to the event history $G$. The columns named $I_{\alpha}$, $I_{\omega}$, $USED_{\alpha}$ and $USED_{\omega}$ shows the event occurrences added to the sets of invalidated initiators, invalidated terminators, used initiators and used terminators for the event type $E_{ab} = E_a \triangle E_b$. The column named $TA \ state$ shows the state of the timed automaton after the event occurrence and the column $TA \ event$ shows the event occurrences in the timed automaton, saved in the timed automaton arrays, that corresponds to the event added to the event history $G$ and that will contribute to the composite event occurrence. As an example the expression $Ea[n]$ shows that it is the $n^{th}$ occurrence of $E_a$ that will be used to form the composite event in the timed automaton.

Each input sequence starts with a row containing the start state of the automaton and its variables. Each row after the start state row corresponds to one event occurrence, and each possible input sequence is separated by a double line. The following is true for counters in the truth tables, where $\mathcal{N}$ is the set of natural numbers:
\[ \forall k, m, n(k, m, n \in \mathcal{N} \land (n > m)) \]

It is also the case that \( t, t_1, t_2, \) and \( t_3 \) represents time of occurrence for the event considered, where \( t < t_1 < t_2 < t_3 \).

**Possible event occurrences in state S0**

Table 5.1 shows the possible permutations of event occurrences of type \( E_a \) and \( E_b \) as the timed automaton pattern is in state S0. The invariant of state S0 states that there are equally many occurrences of type \( E_a, E_b \) and \( E_{ab} \) in the event history \( G \). The invariant of state S0 does not correspond to the conjunction rule for \( E_{ab} \) since there are no occurrences of event type \( E_a \), or \( E_b \) in this state that are not invalidated or used which means that no composite event can be formed as the pattern is in state S0.

<table>
<thead>
<tr>
<th>( G )</th>
<th>( I_a )</th>
<th>( I_\omega )</th>
<th>( USED_a )</th>
<th>( USED_b )</th>
<th>TA state</th>
<th>TA event</th>
<th>( ca )</th>
<th>( a )</th>
<th>( cb )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t^{(1)} )</td>
<td>S0</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( S1 )</td>
<td>( Ea[1] )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( t_b^{(1)} )</td>
<td>S0</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( S2 )</td>
<td>( Eb[1] )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( t^{(n+1)} )</td>
<td>S0</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( S1 )</td>
<td>( Ea[n+1] )</td>
<td>( n+1 )</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>( t_b^{(n+1)} )</td>
<td>S2</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( S2 )</td>
<td>( Eb[n+1] )</td>
<td>( n )</td>
<td>( n )</td>
<td>( n+1 )</td>
</tr>
</tbody>
</table>

Table 5.1: State changes for possible event occurrences as the conjunction pattern in Figure 5.17 is in state S0

\( \mathcal{N} \setminus \{ \emptyset \} \) is true as the timed automaton is in state S0, since there are equally many occurrences of type \( E_a, E_b \) and \( E_{ab} \) as shown by the counters \( ca, cb \), \( a \) and \( b \). This means that there are no occurrences of \( E_a \), or \( E_b \) that has not been already used in a composite event occurrence. The first four rows in the table indicate the start state of the system where
CHAPTER 5. VERIFY TRANSFORMATIONS

there are no event occurrences of type $E_a$ or $E_b$ in the history.

As one occurrence of type $E_a$ is added to the event history, the pattern moves to state S1 and the history of event occurrences satisfies $I N \forall S1$ in the conjunction pattern. The invariant of S1 is true since $ca - a = 1$ and $cb - b = 0$. This means that there is exactly one available occurrence of type $E_a$ in the event history, and no available occurrences of type $E_b$.

In the second case, where there is an occurrence of type $E_b$ in state S0, the pattern moves to state S2 which invariant is also fulfilled since $ca - a = 0$ and $cb - b = 1$.

Possible event occurrences in state S1

Table 5.2 shows the possible event occurrences of type $E_a$ and $E_b$ as the conjunction pattern is in state S1. $I N \forall S1$ states that there are equally many occurrences of type $E_{ab}$ as of type $E_b$ that are not invalidated, or used in the event history $G$ and there are more occurrences of event type $E_a$ than of event type $E_b$ that are not invalidated or used in the history.

$I N \forall S1$ does not correspond to the predicate for the conjunction rule for $E_{ab}$ since there are no occurrences of event type $E_b$ in the event history that are not used or invalidated in this state. Table 5.2 shows that for any occurrences of event type $E_a$ in state S1, the system will remain in state S1, and as an event of type $E_2$ occurs, the system will be transformed to state S3.

Since $n > m$ the invariant of state S1 is true as there are $n$ event occurrences of type $E_a$ and $m$ occurrences of type $E_b$ and $E_{ab}$. The system will remain in state S1 until an event of type $E_b$ occurs. Since there are equally many occurrences of $E_b$ and $E_{ab}$ the precondition for the conjunction rule is not fulfilled in state S1, however, as an event of type $E_b$ occurs, the precondition for the conjunction rule becomes true and the pattern moves to state S3.
### Chapter 5. Verify Transformations

<table>
<thead>
<tr>
<th>G</th>
<th>I_a</th>
<th>I_w</th>
<th>USED_a</th>
<th>USED_o</th>
<th>TA state</th>
<th>TA event</th>
<th>ca</th>
<th>a</th>
<th>cb</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ε_a(n+1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ε_b(n+1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ε_a(n+k)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ε_b(m+1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ε_a(m+1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: State changes for possible event occurrences as the pattern is in state S1

**Possible event occurrences in state S2**

Table 5.3 shows an equal behaviour as table 5.2 with the difference that \(E_b\) is the initiator occurrence and \(E_a\) is the terminator occurrence.

<table>
<thead>
<tr>
<th>G</th>
<th>I_a</th>
<th>I_w</th>
<th>USED_a</th>
<th>USED_o</th>
<th>TA state</th>
<th>TA event</th>
<th>ca</th>
<th>a</th>
<th>cb</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ε_b(n+1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ε_b(n+k)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ε_a(m+1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ε_a(m+1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: State changes for possible event occurrences as the system is in state S2

**Possible event occurrences in state S3**

Table 5.4 shows the possible event occurrences in state S3. As the pattern reaches state S3, the predicate for the conjunction rule is fulfilled. There are one or more occurrences of both \(E_a\) and \(E_b\) in the event history that are not invalidated or used. Depending on if there are more than one unused \(E_a\), or more than one unused \(E_b\) in the event history, the next state will be S0, S1, or S2.

94
### Table 5.4: State changes for possible event occurrences as the system is in state S3

<table>
<thead>
<tr>
<th>$\mathcal{G}$</th>
<th>$\mathcal{I}_a$</th>
<th>$\mathcal{I}_b$</th>
<th>$\mathcal{U}_a$</th>
<th>$\mathcal{U}_b$</th>
<th>TA state</th>
<th>TA event</th>
<th>ca</th>
<th>a</th>
<th>cb</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_{ab}^{13}$</td>
<td>$e_{a(m)}^{11}$</td>
<td>$e_{b(m)}^{12}$</td>
<td>S1</td>
<td>Eab[m]</td>
<td>n</td>
<td>m-1</td>
<td>m</td>
<td>m</td>
<td>m-1</td>
<td></td>
</tr>
<tr>
<td>$e_{ab}^{13}$</td>
<td>$e_{b(m)}^{11}$</td>
<td>$e_{a(m)}^{12}$</td>
<td>S2</td>
<td>Eab[m]</td>
<td>m</td>
<td>m-1</td>
<td>n</td>
<td>m</td>
<td>m-1</td>
<td></td>
</tr>
<tr>
<td>$e_{ab}^{13}$</td>
<td>$e_{a(m)}^{11}$</td>
<td>$e_{b(m)}^{12}$</td>
<td>S0</td>
<td>Eab[m]</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>$e_{ab}^{13}$</td>
<td>$e_{b(m)}^{11}$</td>
<td>$e_{a(m)}^{12}$</td>
<td>S0</td>
<td>Eab[m]</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td></td>
</tr>
</tbody>
</table>

Since $m < n$ the situation in the first example is that there are more than one unused occurrences of type $E_a$ in the event history and there are one unused occurrence of type $E_b$ in the event history. This means that as the assignments of the first unused occurrences of type $E_a$ and $E_b$ to the history $\mathcal{U}_a$ and $\mathcal{U}_b$ are performed, there are one, or more unused occurrences of type $E_a$ in the event history, and no unused occurrences of type $E_b$ in the history. This matches the invariant of state S1 which is the next state of the timed automaton pattern.

The second possibility follows the same reasoning as the first example with the difference that there is one unused occurrence of type $E_a$ and more than one of type $E_b$ so the invariant of state S2 is fulfilled instead.

In the third and fourth example there are exactly one unused event occurrence of both type $E_a$ and of type $E_b$ in the event history, and as they are added to $\mathcal{U}_a$ and $\mathcal{U}_b$ the invariant of state S0 is fulfilled, and the transition between S3 and S0 is taken.

### 5.3.5 Verifying disjunction pattern

In this subsection the equivalence between the disjunction operator and the disjunction pattern presented in section 5.2.2 will be verified. To make it easier for the reader, the
disjunction pattern of the timed automaton shown in Figure 5.13 is reprinted in Figure 5.18.

**Present the disjunction predicate in text and formally**

The operator rule for disjunction is defined by Mellin (2003) to be:

Let $\gamma = \Gamma(E_1 \lor E_2, [\text{start}(\text{span}(\gamma_1), \text{end}(\text{span}(\gamma_1)))])$ in the disjunction rule:

\[
\forall E_1 \lor E_2 \in_m \forall \gamma_1 \in \mathcal{G} : \text{type}(\gamma_1) = E_1 \land \mathcal{X}(E_1 \lor E_2, \gamma_1, \gamma_2, \{E_1, E_2\}, \{E_1, E_2\})
\]

\[
\mathcal{G} \cup = \{\gamma\} \quad \mathcal{U}_s(E_1 \lor E_2) \cup = \{\langle \gamma_1, \gamma \rangle\} \quad \mathcal{U}_u(E_1 \lor E_2) \cup = \{\langle \gamma_1, \gamma \rangle\}
\]

This rule considers the occurrence $\gamma_1$ of type $E_1$, and is equivalent to a rule where the occurrence $\gamma_2$ of type $E_2$ raises the composite event.
CHAPTER 5. VERIFY TRANSFORMATIONS

Formalize variables of the timed automaton pattern

The timed automaton pattern for disjunction does not contain any counters, or other variables that need to be formalized.

Describe invariants of timed automaton states in event algebra

The automaton in Figure 5.18 shows a timed automaton pattern for a composite event of type $E_{ab} = E_a \lor E_b$. Since this is a disjunction, only one of the constituent events must occur for the composite event to be raised.

The set of event types that can initiate or terminate an event occurrence of type $E_{ab}$ is $E_a$ and $E_b$. In other words $\mathcal{E}_\alpha \triangleq \{ E \mid E \in \mathcal{E}_m \land (E = E_a \lor E = E_b) \}$ and $\mathcal{E}_\omega \triangleq \{ E \mid E \in \mathcal{E}_m \land (E = E_a \lor E = E_b) \}$

Invariant in state $S0$:

$\mathcal{IN}_{\mathcal{V S}0} \triangleq \mathcal{AVAILC}_\alpha \cup \mathcal{AVAILC}_\omega = \emptyset$

This means that in state $S0$ there are no prior occurrences of type $E_a$, or $E_b$ that are not invalidated or used.

Invariant in state $S1$:

$\mathcal{IN}_{\mathcal{V S}1} \triangleq (\mathcal{AVAILC}_\alpha = \mathcal{AVAILC}_\omega = \{ \gamma \mid \text{type}(\gamma) = E_a \lor \text{type}(\gamma) = E_b \}) \land$

$(|\mathcal{AVAILC}_\alpha| = 1)$

In other words, if the pattern is in state $S1$, there is exactly one event occurrence of either $E_a$, or $E_b$ in the event history that is not invalidated or used. The disjunction rule is satisfied and a composite event may be formed.
CHAPTER 5. VERIFY TRANSFORMATIONS

Show that all possible inputs for each state corresponds to the state invariants

To verify that the behaviour of the automaton is equal to the behaviour of the derived rule, the state changes caused by sequences of possible event occurrences are formalized and compared. In the following tables, the changes in the event history and timed automaton pattern caused by possible event occurrences in each state are recorded.

Possible event occurrences in state S0

Table 5.5 shows the possible event occurrences and state changes in state S0 for the automaton pattern in Figure 5.18 and a disjunction operator. The state invariant of state S0 is defined to $A V A I L_a \cup A V A I L_w = \emptyset$, and since no event occurrence are used or invalidated in state S0, the automaton must take a transition to state S1 as soon as an event of type $E_a$ or $E_b$ occurs.

<table>
<thead>
<tr>
<th>$G$</th>
<th>$I_a$</th>
<th>$I_w$</th>
<th>$USED_a$</th>
<th>$USED_w$</th>
<th>TA state</th>
<th>TA event</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c^1_a(1)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S0</td>
<td></td>
</tr>
<tr>
<td>$c^1_b(1)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S0</td>
<td></td>
</tr>
<tr>
<td>$c^1_a(n+1)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S0</td>
<td></td>
</tr>
<tr>
<td>$c^1_b(n+1)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S1</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5: State changes for possible event occurrences as the disjunction pattern in Figure 5.18 is in state S0

Possible event occurrences in state S1

Table 5.6 shows the state changes as the automaton in Figure 5.13 is in state S1. The disjunction rule is satisfied and a composite event is raised. The chronicle context predicate
states that the earliest unused and un-invalidated occurrence of type $E_a$ or $E_b$ must be used to compose the composite event. However, each time an event occurs of type $E_a$ or $E_b$, a composite event of type $E_{ab} = E_a \lor E_b$ is raised, so the earliest unused occurrence will also be the only occurrence in the event history.

<table>
<thead>
<tr>
<th>$G$</th>
<th>$I_a$</th>
<th>$I_w$</th>
<th>$USED_a$</th>
<th>$USED_w$</th>
<th>TA state</th>
<th>TA event</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_{ab(n+1)}^f$</td>
<td>$e_a^{l(n+1)}$</td>
<td>$e_b^{l(n+1)}$</td>
<td>$e_a^{l(n+1)}$</td>
<td>$e_b^{l(n+1)}$</td>
<td>S1</td>
<td>$Eab[n+1]$</td>
</tr>
<tr>
<td>$e_{ab(n+1)}^f$</td>
<td>$e_a^{l(n+1)}$</td>
<td>$e_b^{l(n+1)}$</td>
<td>$e_a^{l(n+1)}$</td>
<td>$e_b^{l(n+1)}$</td>
<td>S1</td>
<td>$Eab[n+1]$</td>
</tr>
</tbody>
</table>

Table 5.6: State changes for possible event occurrences as the disjunction pattern in Figure 5.18 is in state S1.

### 5.3.6 Verifying sequence pattern

In the following section, the equivalence between the sequence operator and the sequence pattern presented in section 5.2.3 will be verified. In order to facilitate for the reader, the sequence pattern is reprinted in Figure 5.19 in this section. In the pattern shown in Figure 5.19, the expiration time for the composite event is considered. This means that as there is an event occurrence of the initiator type of the composite event, an event of the terminator type must occur within a specified time period. If the terminator event does not occur within the specified time period, the current initiator event is no longer considered available. According to the definitions in section 5.3.2 expired initiator occurrences does not belong to the set $\mathcal{ALC}_\alpha$ anymore.

**Present the sequence operator predicate in text and formally**

The operator rule for sequence is defined by Mellin (2003) to be as follows:

Let $\gamma \equiv \Gamma(E_1; E_2, [\text{start}(\text{span}(\gamma_1)), \text{end}(\text{span}(\gamma_2))])$ in the sequence rule:
Figure 5.19: Composite event of type $E_{ab} = E_a; \langle expire : 10 \rangle E_b$ in chronicle context with expiration times.

$$\forall E_1; E_2 \in \mathcal{E}, \forall \gamma_1, \gamma_2 \in \mathcal{G} : type(\gamma_1) = E_1 \wedge type(\gamma_2) = E_2 \wedge
$$

$$\mathcal{X}(E_1; E_2, \gamma_1, \gamma_2, \{E_1\}, \{E_2\}) \wedge \gamma_1 < \gamma_2 \wedge \neg(\gamma_1 \parallel \gamma_2)$$

This rule is not sufficient if events occur in parallel, however, it is only the case where events of type $E_1$ and $E_2$ can not occur simultaneously that are considered here.

The precondition for the rule states that for all occurrences of the composite event $E_1; E_2$ that are monitored, there is an event occurrence of type $E_1$ followed by an event occurrence of type $E_2$. The chronicle context predicate makes sure that it is the earliest available initiator and terminator occurrences that are used, and that the composite event
has not expired.

**Formalize variables in the timed automaton pattern**

The automaton in Figure 5.19 contains four counters \(ca, a, cb\) and \(b\). The formal definition for these counters is:

\[
ca \triangleq \{ \gamma_a \mid \gamma_a \in \mathcal{G} \land \text{type}(\gamma_a) = E_a \} \\

\]

\[
cb \triangleq \{ \gamma_b \mid \gamma_b \in \mathcal{G} \land \text{type}(\gamma_b) = E_b \} \\

\]

\[
a \triangleq \{ \gamma_a \mid \gamma_a \in \mathcal{G} \land \text{type}(\gamma_a) = E_a \land (\text{exp}(\gamma_a) \lor \gamma_a \in \text{iot}(\mathcal{U}_a) \lor \gamma_a \in \mathcal{I}_a) \} \\

\]

\[
b \triangleq \{ \gamma_b \mid \gamma_b \in \mathcal{G} \land \text{type}(\gamma_b) = E_b \land (\text{exp}(\gamma_b) \lor \gamma_b \in \text{iot}(\mathcal{U}_b) \lor \gamma_b \in \mathcal{I}_b) \} \\

\]

In other words, \(ca\) is the number of occurred events of type \(E_a\), \(cb\) is the number of occurred events of type \(E_b\). The variable \(a\) represents the number of event occurrences of type \(E_a\) that are used, invalidated, or expired and the variable \(b\) represents the number of event occurrences of type \(E_b\) that are used, invalidated, or expired. This implies that the number of available event occurrences of type \(E_a\) in the event history \(\mathcal{G}\) is \(ca-a\) and the number of available event occurrences of type \(E_b\) is \(cb-b\).

Since event occurrences are consumed, invalidated and expired in chronicle order, the earliest unused occurrence of event \(E_a\) has index \(a+1\), and the earliest available occurrence of type \(E_b\) has index \(b+1\).

**Describe invariants of timed automaton states in event algebra**

The automaton in Figure 5.19 shows a timed automaton pattern for a sequence with expiration times. The proof will be performed for expiration time \(= 10\), however, the proof is assumed to be general enough to hold for all expiration times \(> 0\).

The set of event types that can initiate an event occurrence of type \(E_{ab}\) is \(\{E_a\}\) and
the set of event types that can terminate a composite event of type $E_{ab}$ is \{$E_b$\}. In other words $E_{\alpha} \triangleq \{E \mid E \in E_m \land E = E_{\alpha}\}$ and $E_\omega \triangleq \{E \mid E \in E_m \land E = E_b\}$

Invariant in state S0:
\[ I_{\alpha} S0 \triangleq AVAIL_\alpha \cup AVAIL_\omega = \emptyset \]

In other words, in state S0, there are no event occurrences in the event history of type $E_a$, or $E_b$ that are not used, invalidated, or expired.

\[ IS1 \triangleq (AVAIL_\alpha = \{\gamma_a \mid type(\gamma_a) = E_a \land type(\gamma_a) \in E_\alpha \land \neg exp(\gamma_a)\}) \land\]
\[ |AVAIL_\alpha| = ca - a \land ca - a > 0 \]

Invariant of state S1:
\[ I_{\alpha} S1 \triangleq IS1 \land AVAIL_\omega = \emptyset \]

In other words, in state S1, there are one, or more event occurrences of type $E_a$ that are not used, invalidated or expired. It is also the case that there are no event occurrences of type $E_b$ in the event history that are not used, invalidated or expired.

Invariant of state S2:
\[ I_{\alpha} S2 \triangleq (AVAIL_\alpha = IS1 \land \]
\[ AVAIL_\omega = \{\gamma_b \mid type(\gamma_b) = E_b \land type(\gamma_b) \in E_\omega \land \neg exp(\gamma_b)\} \land |AVAIL_\omega| = cb - b \land cb - b = 1 \land \forall \gamma_a, \gamma_b((\gamma_a \in AVAIL_\alpha \land \gamma_b \in AVAIL_\omega) \rightarrow \gamma_a < \gamma_b) \]

In other words, in state S2, there are one or more event occurrences of type $E_a$ in the event history, and there is exactly one occurrence of type $E_b$ in the event history that are
not used, expired, or invalidated. It is also the case that all un-invalidated, un-expired and unused event occurrences of type $E_a$ precedes the event occurrence of type $E_b$ that are not invalidated, or used.

Show that all possible inputs corresponds to the state invariants

To verify that the behaviour of the timed automaton sequence pattern in Figure 5.19 corresponds to the behaviour of a sequence operator, the state changes that are caused by event occurrences are formalized and compared. In the following tables the following is true $\forall i, k, m, n(i, k, m, n \in \mathcal{N} \land n > m)$. The auxiliary variable $\delta t = t_{Now} - t_{Stamp}$ where $t_{Now}$ is the current system time, and $t_{Stamp}$ is the time when the earliest available initiator event occurred. In the timed automaton pattern, the time stamp for each initiator event occurrence is saved in the array $Apar$ at position $ca$. This means that the timestamp of the earliest available initiator occurrence are saved in the array $Apar$ at position $a+1$, since this is the index of the first occurrence that are not used, invalidated, or expired. As the time expiration $t_{Stamp}$ for the earliest available initiator event occurrence is equal to 10, the initiator is invalidated.

As the time limit expires for a composite event, the initiator occurrence is added to the set of invalidated events. Since the set of expired initiators is not even in the set of $\mathcal{ALL}_\alpha$ this may be an odd way to handle expired event occurrences, but it facilitates the proof table, and it does not change the behaviour of the composite event, or the automaton.

Possible event occurrences in state S0

Table 5.7 shows the possible event occurrences as the sequence pattern in Figure 5.19 is in state S0. The invariant of state S0 states that there are no event occurrences of type $E_a$ or $E_b$ in the event history that are not invalidated, used, or expired. As an event of type $E_b$ occurs in state S0, the event occurrence is immediately invalidated, since it is not possible
to use it as a terminator as there are no unused or un-invalidated initiator occurrences available. Since the event occurrence of type \( E_b \) is immediately invalidated, \( I_N V S_0 \) still holds.

If an event of type \( E_a \) occurs in state S0, the pattern takes the transition to state S1 and an event occurrence of type \( E_a \) is added to the event history \( \mathcal{G} \). This means that there will be one event occurrence of type \( E_a \), and no event occurrences of type \( E_b \) in the event history, that are not used, expired, or invalidated, that correspond to the invariant for state S1.

<table>
<thead>
<tr>
<th>( \mathcal{G} )</th>
<th>( I_a )</th>
<th>( I_w )</th>
<th>( USE _D_a )</th>
<th>( USE _D_b )</th>
<th>( TA ) state</th>
<th>( TA ) event</th>
<th>( ca )</th>
<th>( a )</th>
<th>( cb )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_{a(1)} )</td>
<td>S0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( E_a[1] )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( e_{b(1)} )</td>
<td>S0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( E_b[1] )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( e_{a(n+1)} )</td>
<td>S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( E_a[n+1] )</td>
<td>n</td>
<td>n</td>
<td>k</td>
<td>k</td>
</tr>
<tr>
<td>( e_{b(k+1)} )</td>
<td>S0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( E_b[k+1] )</td>
<td>n</td>
<td>n</td>
<td>k+1</td>
<td>k+1</td>
</tr>
</tbody>
</table>

Table 5.7: State changes for possible event occurrences as the pattern in Figure 5.19 is in state S0

**Possible event occurrences in state S1**

As the sequence pattern in Figure 5.19 is in state S1, there are four possible transitions to take in the timed automaton.

The transition between S1 and S2 is taken as there is an occurrence of type \( E_b \) within the expiration time. As there is an occurrence of type \( E_b \), there will be one event occurrence of type \( E_b \) in the event history and one or more event occurrences of type \( E_a \) that are not invalidated, used or expired. This means that \( I_N V S_2 \) is true. If there is an occurrence of
 CHAPTER 5. VERIFY TRANSFORMATIONS

type $E_a$ in state S1 within the expiration time, the time stamp for this occurrence is saved in the Apar array for later use.

If the time expires for the composite event occurrence as the pattern is in state S1, the pattern takes the transition back to S0 if there are no available occurrences of type $E_a$ in the event history. The initiator event that expired are added to the set of $\mathcal{I}_a$, and hence, the invariant of S0 is fulfilled since there are no unused, unexpired or un-invalidated occurrences of type $E_a$ or $E_b$ in the event history.

If there is available occurrences of type $E_a$ in state S1 as the deadline expires, the previous occurrence of type $E_a$ is invalidated, and the new expiration deadline is set to the occurrence time of the new initiator event plus the expiration time. This means that the invariant of S1 is still true and the pattern remains in state S1.

<table>
<thead>
<tr>
<th>$\mathcal{G}$</th>
<th>$\mathcal{I}_a$</th>
<th>$\mathcal{I}_b$</th>
<th>$USED_a$</th>
<th>$USED_b$</th>
<th>TA state</th>
<th>TA event</th>
<th>ca</th>
<th>a</th>
<th>cb</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>${t_{\text{Now}}, e_{a(n+1)}^{1}}$</td>
<td>S1</td>
<td>n+1</td>
<td>n</td>
<td>k</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\delta t = 10$</td>
<td>S0</td>
<td>n+1</td>
<td>n+1</td>
<td>k</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>${t_{\text{Now}}, e_{a(m+1)}^{0}}$</td>
<td>S1</td>
<td>$Ea[n+1]$</td>
<td>n+1</td>
<td>m</td>
<td>k</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\delta t = 10$</td>
<td>S1</td>
<td>n+1</td>
<td>m+1</td>
<td>k</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mathcal{I}_b(k+1)$</td>
<td>S2</td>
<td>$Eb[k+1]$</td>
<td>n+1</td>
<td>m</td>
<td>k+1</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.8: State changes for possible event occurrences as the pattern is in state S1

**Possible event occurrences in state S2**

In table 5.9 the possible state transitions in state S2 is shown. As the pattern reaches state S2, a composite event is formed. Depending on the number of available initiators, the pattern is transformed to state S0, or state S1.

If there are exactly one available occurrence of type $E_a$ in the event history, the pattern is transformed to state S0. This is because the composite event uses the occurrence of type
CHAPTER 5. VERIFY TRANSFORMATIONS

\( E_a \) and \( E_b \) and as they are added to the set of used events, there are no occurrences of type \( E_a \) or \( E_b \) in the event history that are not used, or invalidated.

If there are more than one available occurrence of type \( E_a \) in the event history, the pattern is transformed to state S1, since there are still unused, un-invalidated and unexpired initiator occurrences in the event history.

| \( \mathcal{G} \) | \( I_n \) | \( I_o \) | USED \(_a\) | USED \(_b\) | TA state | TA event | ca | a | cb | b |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| \( e_{ab}^{(i)} \) | \( e_a^{(i)} \) | \( e_a^{(j)} \) | S2 | \( Eab[i] \) | n | n-1 | m | m-1 |
| \( e_{ab}^{(j)} \) | \( e_a^{(i)} \) | \( e_a^{(j)} \) | S0 | \( Eab[i] \) | n | n | m | m |
| \( e_{ab}^{(j)} \) | \( e_a^{(i)} \) | \( e_a^{(j)} \) | S1 | \( Eab[i] \) | n | n | m | m |

Table 5.9: State changes for possible event occurrences as the pattern is in state S2
Chapter 6

Analysis

This chapter contains an analysis of the results found in the previous chapters; important decisions are discussed as well as weaknesses and strengths of these decisions.

6.1 General transformations

The first objective is to generalize transformations and identify possible limitations of these transformations. The main idea of this objective is to describe transformations found in the previous case study in general terms, to be able to apply them on arbitrary timed automata specifications. The result of this objective makes it easier to see how different constructs of timed automata can be transformed into rules, and highlights problems which must be solved or avoided to be able to apply the method proposed in Ericsson (2002).

6.1.1 Transforming constructs of timed automata from a deterministic environment

The work of generalizing transformations was performed by transforming parts (constructs) of timed automata into rule constructs. Since there may be an infinite set of different
CHAPTER 6. ANALYSIS

timed automaton constructs of varying size, the selected constructs were restricted to a set of small primitives which represents common constructs in timed automata. The constructs were basically taken from a case study performed in a predictable environment (an environment where the order of event occurrences is known a priori) (Ericsson 2002).

The choice of a predictable environment may have affected the selection of which timed automata parts to transform, and it certainly affected the outcome of the transformed rule set. If the occurrence order of events is not known a priori, it must also be ensured that the rule set only reacts on the current event occurrence as the system is in a state which is defined to handle the event. The transformed rule set may for example be forced to behave as the timed automaton specification by using logical events or different modes, restricting the events to only be considered as events as the system is in certain states. Otherwise, the timed automaton can be specified to handle unpredictable event occurrences, e.g. by using the composite event patterns defined in chapter 5.2 in this project.

In section 5.1.4, time constraints specified in timed automata were transformed to the specific syntax of DeeDS (Andler et al. 1995), instead of the more general syntax used in examples not considering time. The reason for using the syntax of DeeDS is that the ability to transform time constraints is highly dependent on the target platform, and DeeDS is an example of a platform that provides powerful ECA semantics even when it comes to expressing time constraints. The examples which did not consider time did only use constructs available in most ADBMS with support for composite events, time constraints on the other hand, are usually implemented as an extension of ECA rules which is specific for different platforms (Eriksson 1998).

The fact that different platforms use different syntax will probably require that a forthcoming implementation of a CASE tool for deriving rules from timed automata is focused on one or a few specific target databases. A standardisation for expressing time constraints as well as for the syntax of ECA rules is of course desirable, but probably not feasible.

108
CHAPTER 6. ANALYSIS

Time constraint specified in timed automata may be transformed into different rule constructs. A clock guard may for example be transformed into a set of rules using time stamps in the condition part, instead of triggering events on a relative time occurrence which was proposed in this project. Further investigation is desirable to identify the most suitable transformation in cases where time constraints in timed automata can be transformed to several different rule constructs. It may for example be the case that one rule construct is more predictable, or more effective, than another and which one to choose depends on the requirements for the current application.

6.1.2 Limitations of transformations

The limitations of transformations between a timed automaton specification and ECA rules are identified to mainly depend on three different behavioural divergences between timed automata and active rules.

I As a new state is reached in a timed automaton, the next transition is eventually taken, nothing is forcing the automaton to take a transition if it is not explicitly specified by invariants or urgent states. In the rule base on the other hand, the evaluation of conditions and execution of actions are depending on the current coupling mode.

II If a guarded transition is evaluated to false, and there are no other valid transitions to take, the automaton waits in the current state until the guard becomes true. In the general case, this behaviour is not trivial to map to ECA rules where a rule is triggered once per associated event occurrence. If the condition is evaluated to false as the event occurs, it will not be evaluated again until the event occurs next time.

III If an external event occurs that is included in the alphabet of the timed automaton, but not in the current state, (the occurrence is not included in the language of the
CHAPTER 6. ANALYSIS

timed automaton) it will pass by unnoticed. However, if an event type is specified for this occurrence in the rule base, this event occurrence will be present in the event history and perhaps trigger a number of rules.

The first divergence is rather easy to overcome if the specification is performed in Uppaal, since it is possible to mark a state as urgent and let the condition be evaluated in an immediate coupling between event and condition. A solution for the second divergence is proposed in section 5.1.6 where guards are reconstructed to event occurrences. The third divergence arises as the order of event occurrences is not exactly known a priori, or if the timed automata specification is wrong (not consistent with the actual order of event occurrences). The problem may preferably be solved by specifying the timed automaton using composite event patterns, or by only allowing rules to be executed in specific modes in the rule set. An event is then only considered as the system is in a state where the event occurrence is accepted by the timed automaton.

Alternative solution for guards

The alternative solution for guards presented in section 5.1.6 makes it possible to transform a guarded transition into rules in the general case. However, the reconstructed model of the guard will be different for different contexts and will also depend on the semantic of the guard, i.e. if the truth value is changed by other processes, if it can be true for different values in different states etc. The algorithm for reconstructing guards presented in section 5.1.6 must be refined until a complete algorithm exists which can perform this reconstructions automatically. The specifier is advised to avoid guards which models states with situations where no valid transition is possible, if the automaton is intended to be transformed into rules.
6.1.3 Prerequisite for transformations

As a timed automaton model is transformed into rules, some prerequisites are identified which must hold for the specified behaviour to be maintained. There must be exactly one valid transition to take every time a timed automaton is in an external state. This implies that if some transition is guarded with a boolean guard, there must be a transition for both the case when the guard is true and when the guard is false. There are two exceptions for this requirement, the first is when a delay is specified with relative event occurrences, and the second exception is for synchronizing transitions. However, at the time when the delay expires, or a synchronisation is possible, there must be exactly one possible transition to take.

6.2 Composite events specified in timed automata

The set of timed automaton patterns specified for composite events is restricted to a set of common operators in recent and chronicle context. Specifying the same operator for different contexts makes the divergence between the same operators behaviour in different contexts obvious. It highlights the importance of considering the target execution model when specifying a system for an active database. Since this dissertation is focused on real-time systems, chronicle and recent were the only contexts considered, however, the operator patterns for cumulative and continuous context are also interesting to specify for comparison reasons.

The set of timed automata operator patterns can be combined to form more complex patterns like $E_{1\text{ seq2}} = E_1; E_2$ $E_{1\text{ seq2 seq3}} = E_1\text{ seq2}; E_3$ or $E = E_{1\text{ seq2}} \triangle E_3$. However, there are still some operators, like a-periodic, non-occurrence etc. which may also be interesting to specify in the future.
6.2.1 Applicability of timed automata patterns

The main aim of specifying composite event occurrences in timed automata is to facilitate the transformation between an arbitrary timed automata specification and ECA rules. This argument is based on the assumption that it is possible to check whether one automaton subsumes another. If the operator pattern and the arbitrary specification can be expressed as a regular expression, the problem of checking if the pattern is included in the timed automaton is well known and easy to solve. If time is considered, this problem is more complex, but still solvable, if the automata can be expressed as timed regular expressions (Alur & Dill 1994). However, the desire to use chronicle context in real time system requires that both counters and clocks is present in the automaton and as far as we know, a practically useful algorithm for finding such patterns in an arbitrary specification remains to be found.

The timed automata patterns for operators can be useful in several ways. Besides the argument that a composite event can be identified in an arbitrary specification, there are a number of other fields of application for the patterns:

I  The operator patterns can be used as a guide when specifying event triggered applications in timed automata. The patterns make it possible to handle events that occur in an arbitrary order and to react on combinations of event occurrences where the occurrence order of the events is not known a priori.

II The patterns may be used as a further development of timed automata CASE tools, where for example the label $E_1 \triangle\langle\text{recent}\rangle E_2$ on a transition implies a conjunction pattern in recent context. The entire pattern does not need to be visible for the CASE tool user, but is used as the tool verifies the specified behaviour.

III As specifications are transformed between timed automata and ECA rules, it is also desirable to reverse engineer models from ECA rules to timed automata. Composite
CHAPTER 6. ANALYSIS

events will be reverse engineered to its corresponding operator pattern.

IV The patterns can be used in a similar way as finite automata are used in Gehani, Jagadish & Shmueli (1992), where finite automata are implemented as a composite event detection mechanism, which can detect composite event occurrences on the fly. However, Gehani et al. (1992) does only consider operators which can be expressed as regular expressions.

V The patterns can serve as a framework for analysing the complexity of composite events in different contexts, for example by using the model checking capabilities in Uppaal.

6.2.2 Specification and implementation issues

The timed automaton patterns assume that one process is monitoring each composite event type. Another alternative is to let each event instance start a new process. This simplifies the automaton patterns since the problem of occurrence order is avoided in the pattern. According to Buchmann, Zimmermann, Blakeley & Wells (1995) it is also preferable as expiration times are considered since the process monitoring the expiring instance simply need to be removed as the deadline for the composite event expires. However, the choice if each instance of a composite event shall be implemented as a separate process is mainly an implementation issue. The main reason for not using one process per event instance in this project is the inability to create new processes dynamically in Uppaal. The behaviour of the implemented rule set is not depending on whether a composite event is specified as a separate process or not, as long the occurrence order of the composite events is kept.

The choice of Uppaal as a specification tool also limits the ability to specify expiration times and order of occurrence. To verify order of occurrence, simulated event parameters were stored in arrays. However, it is not possible to specify dynamic arrays in Uppaal, so
the number of occurred events of each type must be known a priori. It is also the case that clocks cannot be stored in arrays in Uppaal in the way it is used in expiration time patterns in this project. As a workaround, each clock value must be stored in a clock variable on its own in Uppaal (for example four different clock variables instead of one array with the ability to store four clock values).

6.3 Formally verify equivalence of semantics

The usefulness of the operator patterns heavily depends on their correctness. The operator patterns are useless if they do not correspond to the exact behaviour of the composite events in its specified context. This implies that formal verification, or exhaustive testing, is necessary to assure that for all possible input sequences given to the timed automata and the composite event, an equal sequence of outputs is generated by the specifications.

6.3.1 Choice of formal notation

The formal verification can be performed using different kinds of formal notations. Timed automata are a formal specification language, so specifying a composite event in timed automata is a formal specification of the composite event itself. However, it must be verified that the specification corresponds to the composite event pattern for all possible sets of inputs. One of the transformation problems is that the notion of states, which is a central part of a timed automaton, is missing, or at least not explicitly expressed, in a rule set. In this dissertation the state of the rule set is considered to be the set of occurred, used and invalidated events together with the values of event counters.

The notation of the event algebra developed by Mellin (2003) was used to specify invariants of the states using histories of occurred, used and invalidated events. Another approach would be to use a third specification language, for example linear logic or real-
time logic, to express the behaviour in both the composite event in the rule set and the
timed automaton and verify the equivalence of their behaviour using the third language.
However, using a specification language that is developed for expressing composite events
has the advantage that only the behaviour of the timed automaton needs to be defined
in a "new" notation. The opposite approach, for example to describe composite events in
TCTL, a real-time extension to temporal logic (which partly are used as a specification
language for Uppaal), is a third alternative, however, this dissertation is focusing on ECA
rules, so an event description language was the most natural choice.

6.3.2 Equivalence of behaviour

The timed automata operator patterns are verified to have an equivalent behaviour as the
corresponding composite events in the rule base. However, it is only the composition of
events that are considered in this proof. The action part of the rule is an implementation
issue and it must be verified in the testing phase that the implementation of the action is
corresponding to the behaviour of the specification. The patterns are not considered to be
affected by other occurrences in the system than the event occurrences considered in each
pattern since each composite event type is assumed to be monitored in a separate task.

6.4 Related Work

Lots of research has been performed in the area of ECA rules and active databases. How-
ever, many problems still remain unsolved in the general case, since most of the existing
work is focusing of sub-parts of different problems with special constraints or approxima-
tions that must be fulfilled to be able to apply the results. This implies that the complexity
of designing predictable applications using active databases is high, and the ultimate solu-
tion for designing active applications for all different execution models still remains to be
CHAPTER 6. ANALYSIS

found.

The approach proposed in this dissertation is to specify and analyse the application in a high level specification language before the rules are created. Similar ideas are investigated by for example Falkenroth & Törne (1999) where a compiler is built for this purpose and Berndtsson et al. (1997) where ECA rules are derived from speech acts specified in state diagrams. Comai & Tanca (2003) as well as Vaduva, Gatziu & Dittrich (1997) takes the opposite approach as they transform an existing set of rules to an analysable language, focusing on termination and confluence analysis.

The following sections describe the main ideas of different research results, which are identified to be related to this dissertation.

Generating rules from higher level specifications

The problem of constructing a terminating and confluent rule set is addressed by Falkenroth & Törne (1999). They are proposing a compiler approach for generating a set of rules. Constraints on rule sets are expressed in a formal language and the compiler automatically transforms high-level programs into rule sets that conform to the constraints. The approach taken by Falkenroth & Törne (1999) assumes a simplified model where all database states are reachable, and rule actions are executed in a serial way. This project differs from the work of Falkenroth & Törne (1999) since the set of rules will be derived from a formally verified and correct specification. Since the specification is verified to be correct, the rule set is implicitly analysed, if the transformation from timed automaton to ECA rules are preserving the specified behaviour. In a timed automaton it is also possible to specify time constraints, which are characteristic for real time systems.

The work of Berndtsson et al. (1997) is presenting a similar method for transforming rules as the one proposed by Ericsson (2002). In the work by Berndtsson et al. (1997), a method is developed that is transforming the behaviour of a system specified in finite
automata into ECA rules. The guards on the transitions in the finite automaton are mapped to composite events and the activities are mapped to actions. However, the work performed by Berndtsson et al. (1997) is not considering time and it is addressing a rather specific area of applications since its main aim is to model speech acts for cooperating systems.

The IDEA methodology developed by Ceri & Fraternali (1997) is probably the most rigorous methodology for designing database applications with objects and rules. However, the focus of this dissertation, the active rules, is limited to triggers with no composite events in the IDEA methodology.

**Composite events and finite automata**

Composite events are previously identified to be possible to express as regular expressions by Gehani et al. (1992), where finite automata are used for implementing an event monitor. In the work of Gehani et al. (1992), each composite event is identified to correspond to a finite automaton in a similar way as proposed in this project. However, in Gehani et al. (1992) finite automata is used to implement a mechanism that can detect composite event expressions on the fly, the specification of entire applications is not considered. The work of Gehani et al. (1992) does not explicitly name some certain context, but the only context addressed is similar to recent context.

**Confluence and termination analysis**

A lot of work has been done in the area of analysing whether the execution of a set of rules can be guaranteed to terminate, for example Aiken, Hellerstein & Widom (1995), Tschudi, Urban, Dietrich & Karadimce (1997) and Vaduva et al. (1997). However, a complete prediction of the interactions in a set of ECA rules is identified by both (Vaduva 1999, p.83), and Falkenroth (2000) to be an undecidable problem. It is impossible to assert with
CHAPTER 6. ANALYSIS

certainty whether an interaction between rules will take place considering all database states and for all possible action statements. This is why the approach for solving this problem has to contain some restriction, or simplification of the problem, for example to exclude the state of the database in the analysis, or to raise the abstraction of the analysis, as performed in (Vaduva 1999).

The work of Vaduva et al. (1997) is focusing on termination analysis in expressive rule languages. The main idea is that the rules in the rule base are divided into two different sets. The first set contains rules that are dependent on outside event occurrences to be triggered. These rules are identified to be irrelevant for termination analysis because even if they never terminate, it can not be considered as faulty rule behaviour.

The second set of rules is rules which may be triggered by the execution of other rules in the rule set. The aim of the termination analysis is to avoid that rules in the second set is triggering each other indefinitely. For this purpose, a tool is implemented which investigates the characteristics of rule triggering graphs.

The work by Vaduva et al. (1997) is related since it addresses the termination problem for composite events. Lots of work is performed by analysing the behaviour of primitive events, while there are less work on the even more complex behaviour of composite events. However, the work by Vaduva et al. (1997) does only address the correctness of the rule sets behaviour in termination aspects. The issue if the behaviour of the rule set corresponds to its specification is not considered.

The termination and confluence properties are also the focus in the work of Comai & Tanca (2003) where rules are transformed into a generic rule language where an analysis of the desired characteristics are possible. The work of Comai & Tanca (2003) highlights the importance of considering the divergent behaviour in different execution models.

In this dissertation, the confluence and termination characteristics only have an implicit role. The main aim of transforming a timed automaton into a rule set is to gain a
predictable rule set, which includes the termination and confluence properties, however, this dissertation is not explicitly focused on either termination or confluence.

Analysing rules using Petri Nets

The work of Zimmer, Meckenstock & Unland (1996) is addressing the issue of analysing ECA rule sets for non-termination using Petri nets. Zimmer et al. (1996) assumes that each rule in a rule set is individually correct, and explains how to derive a corresponding Petri Net from the rule description. The analysis is then performed on the Petri Net model that is automatically generated from the rule description. The approach of generating a Petri Net model from the rule description takes an opposite direction of what is proposed in this dissertation. According to the authors of Zimmer et al. (1996) their approach has the advantage that there is no additional specification overhead for the developers, since the analysis is performed on the rule description. However, if the rules are directly expressed in formal specification, the rule descriptions are not necessary and can be seen as overhead. In our approach the complex rule sets will be transparent to the application programmer, in a similar way as the machine code is transparent in high-level programming languages.

Petri Nets is also used in the work performed by Schlesinger & Lőrincze (1997) as well as by Xiaou, Medina & Chapa (2002), where Coloured Petri Nets are used for achieving a structural model of the set of ECA rules. The approach taken by both of these works focus on giving a formal definition of rules in a Petri Net model, so each rule in the rule base is also modelled and analysed in the Petri Nets.

Other formal approaches to active rules

Stating properties of events and their relationships formally is performed earlier, for example in the work of Rönn (2001) where a formal comparison between the semantics of D-SNOOP and O-SNOOP is performed when used in the context of cooperating task shar-
CHAPTER 6. ANALYSIS

... protocols. However, the work by Rönn (2001) is not aiming at proving the preserved semantics between specifications of different levels, but on comparing and identifying differences between two similar approaches to event detection.

In the work of Lin, Malec & Nadjm-Tehrani (1997), a rule based language is defined which combines the asynchronous interaction with the environment with synchronous treatment of a response. Lin et al. (1997) claim that by using their rule language, time and concurrency are dealt with in a simpler manner. The semantics of the rule language are formally described and analysed.
Chapter 7

Conclusion

In the following chapter, a summary of the results and the problem will be presented, followed by a discussion about the results, project conclusion and contributions as well as identified future work.

7.1 Project summary

The overall problem addressed in this dissertation is the complexity of developing predictable applications using active databases. The behaviour of active databases is often specified as Event-Condition-Action (ECA) rules which are hard to analyse and maintain since they may interact with each other in many intricate ways. The desire to use active database functionality in real-time systems requires that the rule set has a thoroughly analysed and predictable behaviour, which implies that the issue of analysing a set of rules must be solved.

In this project, the approach for solving the problem of analysing ECA rules is to specify the behaviour of the active application in a high level analysable language, in this case, a timed automaton. As the correctness of the behaviour of the application is verified,
the specification is transformed into rules. For this to be meaningful, the transformation between timed automata and ECA rules must maintain the specified behaviour.

The aim of this project is to verify that transformations between timed automata specifications and ECA rules maintain the specified behaviour. To reach this aim, constructs in timed automata are transformed to ECA rules and their equivalent behaviour is verified. Timed automaton constructs which are hard to transform to ECA rules are identified and alternative solutions are suggested. A set of “operator patterns” are specified in timed automata which represents the behaviour of composite events in recent and chronicle context. The timed automata patterns and their corresponding composite events in chronicle context are formally verified to have equal behaviour.

7.2 Discussion

In this section, strengths and weaknesses of the results concerning the transformations between timed automata and ECA rules are discussed, as well as the choice of high level specification and requirements of the resulting rules set.

7.2.1 High level specification

This entire project is based on the assumption that it is possible to verify the correctness of a system in timed automata. If a timed automaton specification is transformed into rules, possible design flaws will be transformed into the rule set as well. The use of a CASE tool, for example Uppaal, facilitates the verification process, however, some characteristics are still hard to verify in a timed automaton. It is for example hard to automatically verify confluence in a timed automaton, if it is not possible to express and transform priorities on rules. If confluence and termination should be possible to guarantee in the timed automaton specification, it must for example be possible to express if the execution
of an action is changing the value of some condition. However, if conditions can be avoided entirely in the specification, the resulting rule set will only consist of EA rules, and since the proposed method is covering the cases where actions are triggering another event, termination should be possible to guarantee for such systems. However, this statement must be further investigated and removing conditions entirely may be a too high price to pay for guaranteeing termination, since there are other alternatives for solving that problem, for example Aiken et al. (1995), Comai & Tanca (2003) and Vaduva et al. (1997).

The current CASE tools available for timed automata also suffer from capacity limitations. The complexity of timed system analysis depends on the number of clocks and states, and the number of states and clocks possible to analyse in the CASE tools available today is rather limited (Berard & Sierra 2000). Hopefully, this limitation will vanish in the following years due to increased research in verification techniques for timed systems and increased computing capacity. However, although it may not be possible to analyze an entire application in detail using the CASE tools available today, the most important and critical parts of a system is still feasible to verify using these tools.

7.2.2 Transformation issues

The correctness of the generalized transformations presented in chapter 5.1 is not formally verified. However, most of the transformed patterns are very limited in size, so a formal verification is not considered to be necessary for these patterns. It is also the case that the order in which the events occurs are known a priori, which further simplifies the transformation process of the timed automata patterns.

The assumption that the occurrence order is known is true for some classes of systems, e.g. manufacturing cells where every occurrence follows a strict schema. However, there are systems where the occurrence order is not known. The specification process of such systems may be facilitated by the use of composite event patterns, since for example conjunction
patterns accept event occurrences in arbitrary order.

The generality of transformations is mainly focused on transforming an arbitrary timed automata specification to a specific ADBMS. The reason why the transformation can not be performed towards an arbitrary database is that the syntax of the ECA rules are specific for each ADBMS, especially when it comes to specifying time constraints.

The identified application areas for the timed automata operator patterns are presented in section 6.2.1. However, the main aim of the composite event patterns was to be able to find such patterns in an arbitrary timed automaton to facilitate the transformation of composite events. For this to be feasible, an algorithm must be developed which can identify sub-patterns in an arbitrary timed automaton using counters and clocks. Such an algorithm is not necessary in cases where the composite event pattern and the system automaton can be expressed as a regular expression. However, it is not likely that an arbitrary specification specified in Uppaal is possible to express as a regular expression since one of the advantages of Uppaal is the ability to use counters, clocks and arrays in the specification. (We have not succeeded to find an algorithm for this purpose; however, finding such an algorithm was not the aim of this project and is considered as future work.)

One can argue that you can take the acceptance language for a composite event without doing patterns in timed automata first, and this is correct for patterns that are possible to describe as regular expressions, however, if chronicle context is assumed, it is not that easy to express the acceptance language from the set of events. The other part of the study with patterns is that it is possible to express sporadic event occurrences if the automaton is specified using composite event patterns.

7.2.3 Resulting rule set

Applying the method proposed in Ericsson (2002) on a timed automaton also puts requirements on the target ADBMS, which must be able to execute the resulting rule set with
CHAPTER 7. CONCLUSION

acceptable result. The following requirements must be fulfilled by the target execution model:

- Firstly it must be possible to specify composite events. In Ericsson (2002), the conjunction operator is used most frequently, however, the findings in this project implies that the non occurrence operator is required to express the semantics of guards in different context, and the disjunction and sequence operators are also important operators in systems where the occurrence order of events is not known a-priori.

- The contexts addressed in this dissertation are chronicle and recent, since these are the most relevant contexts for real-time systems. Ideally, the target execution models have support for both recent and chronicle context, or in the ultimate case, ability to execute different event types in different contexts, as it is possible to specify in Solicitor (Mellin 2003).

- The coupling mode in the target execution model is preferably detached, to avoid unpredictable execution times for individual transactions.

- Since real-time systems are considered, the target ADBMS must support timely execution of the resulting rules. It must be possible to express deadlines, criticality etc. in the target ADBMS.

The context, coupling mode and time requirements stem from the desire to use the proposed method for real-time systems. A suitable platform which fulfils the stated requirements (except for executing different types in the same application in different contexts) is DeeDS (Andler et al. 1995), which is also used as a target platform in this project where a general ECA rule notation is not sufficient.

The ability to execute different event types in different context within the same application would facilitate the use of timed automata as a specification language, since the
search for patterns could cover all patterns in both recent and chronic context instead of being limited to find patterns for only one of the contexts.

**Resulting rule set**

One experience, also identified in Ericsson (2002) and Berndtsson et al. (1997), is that applying the proposed method for transforming timed automata specifications to ECA rules results in a rule set mainly consisting of EA rules instead of ECA rules.

This means that as a primitive or composite event is detected, the action is executed without the need of checking a condition first. EA rules are simpler than the ECA rules, since they avoid multiple coupling modes and have a simpler semantic, they are also potentially more efficient than ECA rules since no conditions are unnecessarily checked. Using EA rules instead of ECA rules, the focus of the correctness of the rule set must mainly be addressed to the composition of events.

However, even if EA rules are simpler in many ways, expressing the conditions as composite events makes the event part of the rule more complex. According to Berndtsson et al. (1997) the expressive power of EA rules depends heavily on the expressive power of the event language. Not all active databases have support for composite events, and to increase the expressive power further, different consumption policies and logical events are also desirable.

The verification of transformations between timed automata and ECA rules in this project is mainly focusing on the event part of the system. The implementation of the action part of the derived ECA rule must ensure that it corresponds to the state transitions in the timed automata. Even if a formal verification is not performed for the action part, the timed automata specification of the system is still useful since it is possible to derive test cases from the specification which can validate the action part of the rule.
CHAPTER 7. CONCLUSION

7.3 Project conclusion

The aim of this project is to verify the correctness of transformations between timed automata and ECA rules. To reach the aim, three separate objectives are specified which addresses three different issues of transformations; the general case for simple transformations, composite events and formal proof of composite events.

The findings of this project are increasing the knowledge of how to transform timed automata specifications to ECA rules, keeping the specified behaviour during each transformation. This knowledge is crucial in the work of developing a CASE tool for this purpose, which is part of the overall vision for this project.

A CASE tool which allows developers to automatically generate reactive real-time applications from timed automata, giving them access to the power of active databases with remained predictability of applications, is interesting for developers of all kinds of data intensive systems, but especially for developers of systems which needs to respond in a timely manner to external events. Writing applications directly as ECA rules is comparable to writing applications directly in assembler. It is possible, but error prone, time consuming and the systems will be hard to debug and analyse. Performing the specification part in a higher level language makes it possible for the developer to focus on the correctness of system requirements, ensuring that the right system is built, instead of low level implementation issues.

The high level specification must not necessarily be timed automata as suggested in this project. However, even if limitations are identified, which must be avoided in specifications of applications, or rebuilt in the transformation phase, the specification power of timed automata is still attractive to use for this purpose due to its high analysability capability. It is also the case that even if there are divergences in the semantics of timed automata and ECA rules when it comes to guards and conditions, there are also several similarities.
CHAPTER 7. CONCLUSION

which makes it worth the effort of continuing this work using timed automata.

The aim and objectives of this project are considered to be fulfilled. The first objective generalises transformations and identifies constructs in timed automata that are hard to transform in the general case. In the second objective, the behaviour of composite event operators are specified as timed automata patterns and in the third objective these patterns are formally verified for the chronic context.

7.4 Contributions

The overall contribution of this project is an increased set of verified transformations between timed automata and ECA rules. This overall contribution can be separated into the following more specific contributions:

- A structured set of general timed automata constructs and their corresponding rules are presented for transformations in predictable environments.

- Limitations of transformations between timed automata and ECA rules when using the method proposed in Ericsson (2002) are identified. We have also presented suggestions for how to solve the limitations caused by guards for some cases and an outline for an algorithm serving as a first step towards a general solution for the problem with guards.

- The semantics of operators are specified in timed automata for recent and chronic context. Several applications areas for using the timed automata patterns are identified beside the original aim of facilitating the identification of composite events in timed automata.

- Formal proofs of behavioural equivalence between time automata patterns and composite events are performed.

128
• Future work are identified both in the area of continuing development of a CASE tool for generating rule sets and in the area of enhancing the specification power in timed automata CASE tools.

7.5 Future work

This project is defined to be one step forward in the process of developing a CASE tool for generating an analysed and predictable set of ECA rules, so the most obvious future work is to take another step toward this high level aim. The next step might for example be to complete verifications of composite events, construct specific algorithms for transforming guards, to gather requirements for the forthcoming CASE tool or to investigate existing similar CASE tools and their functionality. In the following, the most interesting future works identified in this project are summarised.

Specific algorithm for guards

As already pointed out, the ability to transform guarded transitions is limited in the general case. Some alternative solutions are suggested in this project where the behaviour of the guard is modelled in a separate automaton. If a new system is specified, guards can be avoided in the specification and the alternative specification solutions for guards may serve as a guide for reconstructing transitions for which guards are needed.

However, if an existing system, using guards, is to be transformed into rules, an algorithm for automatically reconstructing the behaviour of guards is needed. The algorithm must search the model for all transitions that is either changing the value of a guard or using a guard on a transition. A new automaton, modelling the behaviour of the guard must be built based on the search result.
CHAPTER 7. CONCLUSION

In this project, an outline of an algorithm for reconstructing guards was suggested. However, a more detailed description of the algorithm is necessary and the result of using this algorithm must be validated.

Specify and verify additional operators

In this project, the timed automaton patterns for operators were specified conjunction, sequence and disjunction in chronicle and recent context and formally verified for chronicle context. It is interesting to extend the set of specified and verified operators with all useful operators in different contexts.

Algorithm for finding timed automata patterns

As far as we know, there is no algorithm for finding patterns of timed automata that are not possible to express as timed regular language, in an arbitrary specification. This problem is identified by Alur & Dill (1994) to be unsolvable, however, it might be solvable if for example other constraints are put on the specifications which decrease the search space.

Enhancing specification power of timed automata CASE tool

Since the operator patterns are assumed to facilitate the specification of systems with sporadic events, it is interesting to investigate the possibility of integrating composite events in a CASE tool, based on timed automata but extended with ECA rule features. In such CASE tools, the specifier could type in the event types, operator and context on a transition instead of specifying the entire operator pattern. This will facilitate the work of specifying event triggered systems with sporadic events in timed automata.
Acknowledgement

I want to thank my great guru Robert Nilsson for supporting me and giving me feedback during this entire project and Jonas Mellin for helping me in the starting phase and for valuable comments. I also want to thank Prof. Sten F. Andler for advice and feedback and Dr. Mikael Berndtsson for pointers to interesting literature and comments on my report. Thanks to all people in the DRTS group at Högskolan i Skövde for your feedback on my presentations, especially to Birgitta Lindström for reading this dissertation and giving me great feedback.

I also want to thank my fellow students Hanna and Jane for giving me lots of inspiration and motivation and my family Andreas, Martin and Philip for making my life easier and backing me up.
Bibliography


BIBLIOGRAPHY


134


Salling, L. (1998), *Formella språk, automater och beräkningar*, Eklundshofs Grafiska AB.


BIBLIOGRAPHY

