Mälardalen University Press Dissertations
No. 44

Event Pattern Detection for Embedded Systems

Jan Carlson

2007

Department of Computer Science and Electronics
Mälardalen University

ISSN 1651-4238
ISBN 978-91-85485-48-2
Printed by Arkitektkopia, Västerås, Sweden

Mälardalen University Press Dissertations
No. 44

EVENT PATTERN DETECTION FOR EMBEDDED SYSTEMS

Jan Carlson

Akademisk avhandling

som för avläggande av Filosofie doktorsexamen i Datavetenskap vid Institutionen för datavetenskap och elektronik kommer att offentligen försvaras fredagen, 15:e juni, 2007, 14.00 i Gamma, Hus U, Högskoleplan 1, Rosenhill, Västerås.

Fakultetsopponent: Professor Kim Larsen, Institut for Datalogi, Aalborg Universitet, Danmark.

Institutionen för datavetenskap och elektronik
Mälardalen University, Box 883, SE-72123 Västerås

Abstract

Events play an important role in many computer systems, from small reactive embedded applications to large distributed systems. Many applications react to events generated by a graphical user interface or by external sensors that monitor the system environment, and other systems use events for communication and synchronisation between independent subsystems. In some applications, however, individual event occurrences are not the main point of concern. Instead, the system should respond to certain event patterns, such as "the start button being pushed, followed by a temperature alarm within two seconds". One way to specify such event patterns is by means of an event algebra with operators for combining the simple events of a system into specifications of complex patterns.

This thesis presents an event algebra with two important characteristics. First, it complies with a number of algebraic laws, which shows that the algebra operators behave as expected. Second, any pattern represented by an expression in this algebra can be efficiently detected with bounded resources in terms of memory and time, which is particularly important when event pattern detection is used in embedded systems, where resource efficiency and predictability are crucial.

In addition to the formal algebra semantics and an efficient detection algorithm, the thesis describes how event pattern detection can be used in real-time systems without support from the underlying operating system, and presents schedulability theory for such systems. It also describes how the event algebra can be combined with a component model for embedded system, to support high level design of systems that react to event patterns.

ISSN 1651-4238

ISBN 978-91-85485-48-2

Abstract

Events play an important role in many computer systems, from small reactive embedded applications to large distributed systems. Many applications react to events generated by a graphical user interface or by external sensors that monitor the system environment, and other systems use events for communication and synchronisation between independent subsystems. In some applications, however, individual event occurrences are not the main point of concern. Instead, the system should respond to certain event patterns, such as "the start button being pushed, followed by a temperature alarm within two seconds". One way to specify such event patterns is by means of an event algebra with operators for combining the simple events of a system into specifications of complex patterns.

This thesis presents an event algebra with two important characteristics. First, it complies with a number of algebraic laws, which shows that the algebra operators behave as expected. Second, any pattern represented by an expression in this algebra can be efficiently detected with bounded resources in terms of memory and time, which is particularly important when event pattern detection is used in embedded systems, where resource efficiency and predictability are crucial.

In addition to the formal algebra semantics and an efficient detection algorithm, the thesis describes how event pattern detection can be used in real-time systems without support from the underlying operating system, and presents schedulability theory for such systems. It also describes how the event algebra can be combined with a component model for embedded system, to support high level design of systems that react to event patterns.

Acknowledgements

First and foremost, I want to thank my supervisor, Professor Björn Lisper, for supporting me during my PhD studies, and in particular for having patience with my repeated excursions outside the scope of the original project plan.

For proof-reading various parts of the thesis, and for comments and discussions (especially on some particularly elusive aspects of real-time scheduling), well deserved credit goes to Radu Dobrin, Kaj Hänninen, Jukka Mäki-Turja and Jonas Mellin.

Also, my warmest thanks go to past and present colleagues at the department, for a friendly atmosphere and for long and pointless coffee break discussions: Nerina Bermudo, Markus Bohlin, Waldemar Kocjan, Johan Lindhult, Andreas Sjögren, Xavier Vera and all the rest.

I am also grateful to the people I have collaborated with over the years. How come other people's research always seems much more interesting? Those that have not been mentioned elsewhere on this page are, roughly in order of appearance, Pawel Pietrzak, Gerhard Fohler, Tomas Lennvall, Mikael Åkerholm, Hans Hansson, Paul Pettersson, John Håkansson, Thomas Nolte, Massimo Tivoli, Mikael Nolin, Ivica Crnkovic and Rikard Land.

Finally, I am truly indebted to my family for their love, encouragement and support: Marina, Nell, Em, Bengt, Kerstin, Lena and Peter.

Jan Carlson
Västerås, May, 2007

This research was funded in part by CUGS (the National Graduate School in Computer Science, Sweden).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The concept of *events* appears in many different forms in different areas of computer system design and implementation. For example, interrupt events indicate that something has happened in the environment that the system might want to react to, for example a sensor update or a key being pressed. On a higher level, large systems can be designed according to an event-based architectural style, meaning that the communication between different parts of the system is based on a publish/subscribe interaction paradigm, where event consumers that have expressed an interest in a certain event is notified whenever a matching event occurrence is published [50]. Events can also be also useful when monitoring complex systems, such as telecommunication networks [48], air traffic [91] and stock markets [62].

In some applications, the desired behaviour is related to complex patterns of events rather than to single event occurrences. A systematic way to handle this is to separate the detection of such event patterns from the implementation of the appropriate reactions. The detection mechanism processes the simple events of the system and matches them against the patterns that are of interest. When a full match is detected, this is announced to the rest of the system, where pattern occurrences can be used in the same way as occurrences of simple events, for example to trigger a response or to modify an internal state. This separation of concerns facilitates design and analysis, since the specification of complex event patterns can be given a formal semantics independent from the application in which it is used, and the other parts of the system are free

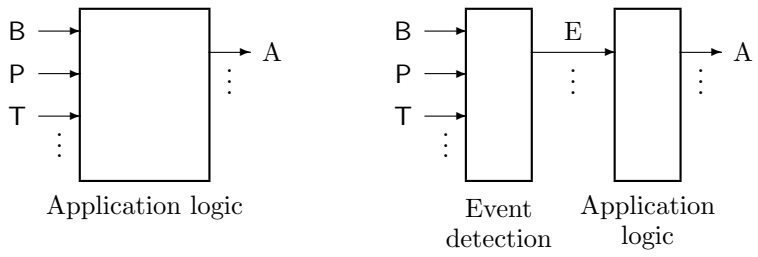

Figure 1.1: Integrated (left) and dedicated (right) detection of event patterns.

from auxiliary detection rules and information about partially completed patterns.

As an example, consider a system where the external events include a button B, a pressure alarm P and a temperature alarm T, where one desired reaction is that the system should perform a particular action A when the button is pressed twice within two seconds, unless either of the alarms occurs in between. This could, for example, be achieved by a collection of rules that specify reactions to the three primitive events, so that the combined behaviour implements the desired reaction. Alternatively, a pattern E can be defined that corresponds to the described situation, in which case the desired reaction can be achieved by a single rule stating that an occurrence of E should trigger the action A. The two approaches are illustrated in Figure 1.1.

In some related work, the term *composite event* is used for what is called *event pattern* in this thesis, emphasising the fact that the pattern is viewed, and used, as an event in the rest of the system.

1.1 Objectives and motivation

A number of methods have been proposed to specify event patterns in different settings. Some frameworks use regular expressions [61] or finite automata [122, 160, 72], and other use some variant of modal or temporal logic with explicit support for events [139, 38, 111]. Another technique is event algebras, where event patterns are defined by expressions built from simple events and algebra operators. This approach is common in

languages for active databases [35, 109, 161], but also in some general, high-level event notification systems [65, 76, 157].

Naturally, the nature of the domain in which event pattern detection is used determines the criteria against which a particular technique should be evaluated. This thesis focuses on the area of embedded real-time systems, where low and predictable resource usage is crucial [28]. These applications typically require that bounds for memory usage and execution time can be determined statically, in order to guarantee timely responses. Furthermore, they often appear in safety-critical applications, meaning that failure or malfunction may result in serious damage to people, equipment or the environment [144]. Because of this, traditional testing is often complemented by some type of formal verification, which requires that the semantics of the pattern specification technique is formally defined.

Concretely, the main goal of this project has been the development of a declarative event algebra that (i) complies with algebraic laws that intuitively ought to hold for the operators, and (ii) permits an efficient detection mechanism for which resource bounds can be determined statically. To the best of our knowledge, there exists no event pattern detection technique where both these aspects are addressed satisfactory. In isolation, a bounded implementation is straightforward to achieve, for example based on finite automata, but these methods are typically not able to detect pattern instances that overlap in time, or suffer from other peculiarities (see Section 3.2 for examples). Similarly, there are highly expressive languages with simple and intuitive semantics, for which no bounded detection method exists in the general case.

We also wanted to investigate some aspects that are specific for event pattern detection in the context of embedded systems. These include high-level design support for embedded systems that react to event patterns, how event pattern detection can be used in a real-time system without direct support from the underlying operating system, and how event pattern detection affects scheduling and schedulability.

1.2 The approach

Table 1.1 lists the operators of the proposed algebra, and describes their meaning informally. For formal definitions, see Section 4.2. As an example, the pattern “the button (B) is pressed twice within two seconds,

Table 1.1: Informal description of the algebra operators.

Operator	Notation	Informal meaning
Disjunction	$A \vee B$	A or B (or both) occurs.
Conjunction	$A+B$	A and B have occurred (in any order, and possibly not simultaneously).
Sequence	$A;B$	An occurrence of A followed by an occurrence of B.
Negation	$A-B$	An occurrence of A, during which B does not occur.
Temporal restriction	A_τ	An occurrence of A shorter than τ time units.

unless either of the alarms (P or T) occurs in between", discussed above, can be specified by the expression $(\mathsf{B};\mathsf{B})_{2\ \mathrm{sec}} - (\mathsf{P} \vee \mathsf{T})$. These operators, or variants of them, are found in many of the existing event algebras from different application domains [35, 65, 76, 161], but the use of interval based semantics (see Section 2.4.8) allows more general constructs for negation and temporal restriction, compared to many other formalisms.

The algebra is defined by a set-based declarative semantics, rather than in terms of state automata, Petri nets or similar. This simplifies the tasks of proving algebraic properties, at the cost of not providing a direct model of how the algebra can be implemented. Instead, we provide a separate imperative detection algorithm to investigate time and memory usage in detail, and verify that this algorithm correctly implements the declarative algebra semantics.

To preserve intuitive operator properties under operator composition, techniques such as interval based semantics are used, and a carefully designed restriction policy is applied to handle the memory complexity caused by some of the properties. These techniques are described further in Sections 2.4.8 and 4.2.

To the user of this algebra, the impact of the restriction policy is that at any time when there are one or more occurrences of a pattern according to the operator semantics defined informally in Table 1.1, exactly one of them will be detected.

1.3 Contributions

The main contributions of the thesis are summarised below. A more detailed presentation of the contributions is given in Section 8.1, followed by a discussion on how these contributions relate to existing work in the area.

- A novel declarative event algebra with well-defined algebraic properties that intuitively ought to hold for the algebra operators. These properties facilitate formal as well as informal reasoning about the algebra and the behaviour of a system that uses it.
- A detection algorithm for the algebra that correctly detects any expression with bounded memory. The algorithm is formally verified with respect to correctness and complexity.
- A task model for real-time systems where some tasks are triggered by event patterns, schedulability analysis techniques for this task model, and a strategy for realising pattern triggered tasks without support from the operating system.
- An extension of a software component framework for embedded systems, which allows components to be triggered by complex event patterns.

1.4 Related publications

The event algebra has evolved into the current form through a number of preliminary versions. Below are listed the publications that relate to the work presented in this thesis. A complete publication list can be found in Appendix D.

- **An Interval-Based Algebra for Restricted Event Detection**, Jan Carlson and Björn Lisper, In *Proceedings of the First International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS 2003)*, Marseille, France, September 2003.

  This paper presents a first version of the algebra. The temporal restriction construct is not present, and two different restriction policies are used (one for sequences and one for the remaining operators). No general resource bounds are given, and the algebraic

properties are weak compared to the current version, in particular the relation between the unrestricted semantics and the result when the restriction policy is applied.

- **An Improved Algebra for Restricted Event Detection**, Jan Carlson and Björn Lisper. *MRTC Technical Report MDH-MRTC-159/2004-1-SE*, February 2004.

  Here, the algebra is extended with a temporal restriction, but for sequence constructs only. The main result is that detection with limited memory is ensured for expressions where every sequence has a finite temporal restriction.

- **An Intuitive and Resource-Efficient Event Detection Algebra**, Jan Carlson. *Licentiate thesis No. 29*, Mälardalen University, ISBN 91-88834-49-2, June 2004.

  The licentiate thesis contains the current version of the algebra semantics, but the detection mechanism is significantly weaker than later versions. Bounded memory can be ensured only for a subset of expressions, and the memory footprint depends on the time restrictions used in the expression, and on the minimum interarrival time of primitive events. The licentiate thesis also presents an optimisation algorithm that transforms an expression into an equivalent form that can be more efficiently detected, and a prototype Java implementation.

- **An Event Detection Algebra for Reactive Systems**, Jan Carlson and Björn Lisper. In *Proceedings of the fourth ACM International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, September 2004.

  This paper presents the algebra and the detection algorithm from the licentiate thesis.

- **An Event Detection Algebra for Reactive Systems**, Jan Carlson and Björn Lisper. *MRTC Technical Report MDH-MRTC-117/2004-1-SE*, April 2004.

  This technical report extends the previous paper with formal proofs.

- **An Event Algebra Extension of the Triggering Mechanism in a Component Model for Embedded Systems**, Jan Carlson and Mikael Åkerholm. In *Proceedings of the Workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA)*, Edinburgh, Scotland, April, 2005.

  The paper explains how the event algebra can be used to extend the triggering mechanism of SaveCCM, a component model for embedded systems.

1.5 Thesis outline

Chapter 2 gives an overview of the application domains where events and event patterns are commonly found, and introduces basic terms and concepts related to event pattern detection. In particular, the chapter discusses qualities and properties that are typically desired in a mechanism for event pattern detection, as well as techniques that can be used to achieve these goals. Chapter 3 surveys related work, describing a number of approaches to handle event pattern specification and detection in different domains.

The syntax and semantics of the proposed event algebra is given in detail in Chapter 4, together with a description of algebraic laws and other significant properties. Chapter 5 presents the detection algorithm and gives a formal proof that this algorithm correctly implements the algebra semantics from the previous chapter. The algorithm is analysed with respect to complexity, and experimental results are presented to give a more detailed view of the resource requirements.

Chapter 6 addresses how event pattern detection can be incorporated in a real-time system to support activities that should only be performed in response to certain event patterns. We show how this can be achieved without support from the underlying operating system, and present schedulability theory for the proposed task model. In Chapter 7, the event algebra is combined with SaveCCM, a component model for embedded systems, to support high level design of systems that react to event patterns.

Finally, Chapter 8 concludes the thesis by highlighting the main contributions and comparing them with existing work in the area. A number of possible future research directions are also described.

Chapter 2

Background

This chapter provides a general overview of how events and event pattern detection are used in computer systems, and more detailed discussions on issues that are closely related to the work in this thesis.

2.1 Events and event patterns

In a very general sense, an event represents a particular type of action or change that is of interest to the system, occurring either internally within the system or externally in the environment with which the system interacts. We refer to one particular case of such an action or change as an *event occurrence* (or, in the formal definitions, an *event instance*). For example, consider the event MouseClick in a graphical user interface. In a particular runtime scenario there will be a number of occurrences of this event, one for each time the user clicks the mouse button. In general, events are considered to be *recurrent*, meaning that an event can occur several times, although there are certain events that have exactly one occurrence, such as a calendar event or a SystemStartup event.

In some event based applications, individual event occurrences are not the main point of concern, but rather the occurrences of certain event patterns. To support this, an event framework can provide means to specify event patterns, allowing these patterns to be used in the same way as ordinary events. To the rest of the system, the pattern is viewed as a *composite event* (sometimes called a *complex* or *compound* event) that

occurs whenever the *primitive* events of the system occur in accordance with the pattern.

Moving the details of pattern detection from the application to the event framework provides a clear separation of concerns between the mechanisms of pattern detection on one hand, and the definition of how to respond to them on the other. This separation has a number of possible benefits, including the following:

- Patterns are explicitly defined, rather than being an implicit consequence of how the responses are defined. This means that the patterns are available for analysis, transformation, formal and informal reasoning, etc. It might, for example, be possible to determine automatically how often a particular pattern can occur, which can be of great importance if timely responses have to be guaranteed.

- In a case where several parts of the system are interested in the same pattern, or patterns with some common parts, it can be more efficient to perform the overlapping parts in one place, instead of letting each part perform the detection locally.

- The number of event occurrences that are sent between nodes in a distributed system can sometimes be significantly decreased. This is of particular importance in systems where the bandwidth between where an event occurs and where it is to be handled, is limited. If pattern detection can be performed close to the event source, the network load can be reduced.

- If the event framework provides a simple and intuitive way to specify event patterns, it may be easier for the developer to quickly achieve the desired behaviour.

2.2 Application domains

For the following discussion, we have identified three categories of application domains where events and event pattern detection are found, namely “reactive systems”, “event based communication” and “ monitoring and event data mining”. These categories are neither disjoint nor covering all aspects of events, but serve to give some structure to the presentation.

2.2.1 Reactive systems

In some systems, events are the main means by which execution is driven. A *reactive* system is designed to remain inactive until triggered by an occurrence from a particular set of events. This category includes embedded systems where hardware interrupts from sensors cause tasks or transactions to be released for execution, carrying out the proper response to that particular event.

Non-embedded software is also sometimes designed with an event driven execution model. In particular, many graphical user interfaces generate events according to user actions such as moving the mouse over a particular object or clicking a button, and each event triggers the execution of some associated code [14, 118]. It can also be the case that some parts of the system follow a traditional style of control, and other parts are reactive. For example, in a simulation application or a computer game designed according to the *model-view-control* design pattern [56], the model subsystem typically executes continuously, but it is affected by user interaction events in the reactive control subsystem.

Visual Basic [155] and JavaScript [110] are examples of programming languages that are often used in reactive, event driven applications where the overall flow of the program is determined by user actions, rather than by explicit control structures in the program. Other programming languages support event-based communication and control transfer as a complement to the ordinary language mechanisms, either as an integrated part of the language, e.g., in Java [52], C# [135] and Tcl [152], or through libraries or extensions such as libevent [124] and liboop [49].

Finally, active databases should be mentioned as an example from this category, since the algebra presented in this thesis has much in common with work in this area. In an active database the functionality of an ordinary, passive database is extended with means to define reactions to situations that arise within or outside the database [120]. The reactions are often specified by so called *event-condition-action rules* (ECA rules) stating that when a certain event occurs, the action should be performed if the condition is satisfied. As an example, a staff database could contain a ECA rule stating that when the salary table is updated, and if the new value is more than 10% higher than the old value, a notification should be sent to the management. To further increase the flexibility, many active databases allow rules that are triggered by event patterns specified, for example, by an event algebra [35, 57, 60].

2.2.2 Event-based communication

Events can also be used for communication within a system, as a complement to message passing, remote procedure calls, etc. The *publish/subscribe* interaction paradigm [50, 51] allows components or subsystems to indicate their interest in receiving notifications about occurrences of a particular event by registering a subscription, either directly to the event producer (as in the *observer pattern* [56]) or to an intermediary subscription engine. When an event occurrence is published, it is matched against the current subscriptions and distributed to the appropriate subscribers (see Figure 2.1). Subscription can either be *topic based*, i.e., a component subscribes to occurrences of particular events, or *content based*, in which subscribers are notified whenever the contents of a published event occurrence match the constraints of the subscription [50].

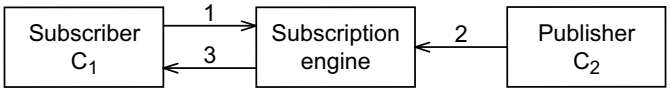

Figure 2.1: Example of the publish/subscribe interaction paradigm. Component C_1 registers a subscription with the subscription engine (1). When component C_2 publishes an occurrence that matches the subscription (2), the occurrence is forwarded to C_1(3).

An important advantage of this communication style is that communicating entities need not be aware of each other, and can communicate asynchronously without blocking. Furthermore, they need not be physically connected, or active at the same time, in order for the communication to take place, since the subscription engine can delay the notification of subscribers that are temporarily unavailable without affecting the event producer. This spatial and temporal decoupling is particularly suitable for large, heterogeneous and distributed systems, since it allows parts of the system to be modified or replaced, possibly even dynamically during runtime, without having to modify the rest of the system [51].

Some publish/subscribe frameworks support subscription to event patterns as well as to single events [90, 122]. For example, a subsystem might subscribe to the pattern *"A and B occur within 2 seconds"*, instead

of subscribing to the individual events A and B and then detect the desired situation internally.

Event related services can be supplied by the operating system, by a programming language or environment, or by an event based middleware. Middleware, i.e., software located between the operating system and the applications [87], facilitates the design of complex, distributed systems by hiding low-level details related to distribution and the underlying operating system and hardware. Event based middleware, e.g., Hermes [122] and READY [65], provide a uniform high-level interface of event related services, which makes it easier for heterogeneous subsystems to communicate seamlessly.

2.2.3 Monitoring and event data mining

Events are also used for managing, monitoring or exploring complex systems, including software systems [104] or networks [22] but also real-world systems like stock markets [62]. Contrasting the first category, these systems are typically not reactive in the sense that they become active only in response to event occurrences. Rather, they look for trends, patterns and relationships in event data, either with respect to a given description (for example monitoring the correctness of a system by matching output events to a specification), or establishing new, statistically valid, correlations as in the work of Padmanabhan and Tuzhilin [117]. Here, a main concern is dealing effectively with very large volumes of event occurrences, and to filter out only those that are of interest in a particular situation.

Work that falls into this category includes monitoring of real-time systems [104], supervision of telecommunication networks [48] and air traffic control [91]. Additional examples are network monitoring [23] and specification of event based security properties [25].

2.3 Primitive events

The primitive events can originate from within the system, as is the case for event based communication between subsystems, or from external devices such as sensors. It is also possible that an external real-world action causes an internal primitive event to occur, e.g., clicking the mouse button might result in an occurrence of the WindowClosed event or

the MenueItemSelected event, depending on the current location of the mouse pointer.

Monitoring an external system can either be based on the visible input- and output events, or with access to internal events in the underlying architecture, such as timeouts, communication, synchronisation events, exceptions, etc. Another possibility is to modify the existing system to generate specific monitoring events that provides sufficient information about the internal processes to allow adequate monitoring.

2.3.1 Ordering and timestamping

For event pattern detection, the relative order of primitive event occurrences is clearly significant. Typically, the framework must be able to handle the fact that occurrences sometimes are reported at a faster rate than they are processed by the detection mechanism. This happens in particular if pattern detection is performed periodically at predetermined points in time, or when detection can be delayed by more critical activities, but it can also be that several primitive events occur during the processing of an earlier occurrence. A straightforward solution to uphold the occurrence order is to store primitive occurrences in a queue, sometimes called the *event history* [34], as soon as they are reported. When invoked, the pattern detection mechanism processes the queued occurrences in FIFO (first-in-first-out) order, either one occurrence at a time, or all at once but taking the order into account when processing them.

Systems that are distributed over multiple computational nodes require some additional efforts. In particular, primitive occurrences can no longer be inserted into a global queue directly when they occur, since detection might not take place on the node where the queue is located. A common technique is to include an explicit timestamp in the representation of an event occurrence, which allows the event history to be ordered based on occurrence time rather than the order in which occurrences are inserted. For external events, the timestamp can either represent the time when the occurrence entered the system, or it can be set by the external event source [142]. Timestamping of primitive occurrences is also required if the formalism allows patterns that depend explicitly on absolute occurrence times or the relative time between several occurrences, e.g., *"A followed by B within 2 seconds"*.

In order to ensure that occurrences are processed in the same or-

der in which they occurred, processing must be delayed until the first occurrence in the queue is older than the maximum time it can take to distribute information about a primitive occurrence to the event history [142]. Alternatively, occurrences can be processed as soon as they arrive, if a roll-back mechanism can be invoked should an occurrence with earlier timestamp arrive later.

In a distributed setting, one must also consider issues such as clock drifting, i.e., that the clocks on different nodes might drift over the system lifetime. This means that if two events occur on different nodes, the order of their timestamps might differ from the order in which they actually occurred. This is generally handled by some clock synchronisation technique that bounds the maximum difference between two clocks at any given time. For details on clock synchronisation, see Tanenbaum [146] and the survey by Ramanathan et al. [127]. Detection of event patterns in distributed systems has been addressed by, e.g., Schwiderski [136], Liebig et al. [92] and Pietzuch [122].

2.3.2 Event parameters

In addition to time, many primitive events have specific information associated with each occurrence, to further specify the nature of the action or condition that caused the event to occur. This additional information is often referred to as *event parameters* [136]. For example, a HighTemperature event raised by an external sensor might include the measured temperature, a ButtonPressed event in a graphical user interface typically carry a reference to the particular button that was pressed, and an internal communication event occurrence can contain parameters similar to those of a function call.

2.3.3 Hierarchical events

Primitive events can be hierarchically structured, meaning that a group of events are categorised under a common name, and this name is also seen as an event. A subscriber can chose to subscribe either to a specific event in the group or to the general event, in which case it will receive the occurrences of all events in that group. For example, Figure 2.2 depicts parts of the event hierarchy of a graphical user interface. In this scenario, a subscription to the pattern *"A* ButtonPressed *event followed by a* Mouse *event"*, would match both the case of the button

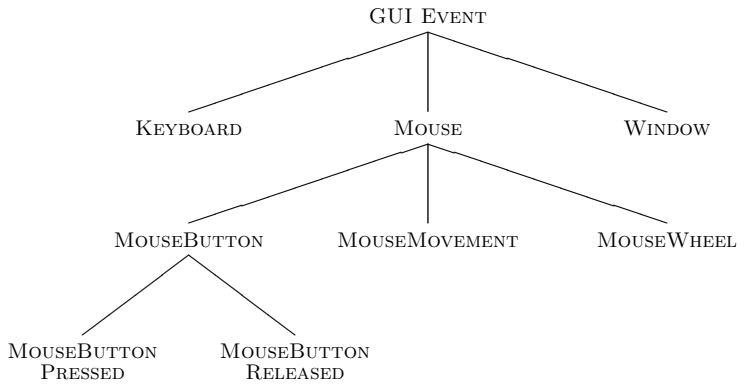

Figure 2.2: Parts of a possible GUI event hierarchy.

being pressed and released, or pressing the button and then moving the mouse.

Another example of hierarchically structured events, is presented by Bernauer et al. [20]. They describe an event algebra suited for XML document events, such as modification of attributes, or the insertion and deletion of nodes. The hierarchical structure of XML is reflected in the primitive events, so that the modification of a particular node in the XML document is considered as a modification event on that node, but also on each enclosing node. For example, if the XML document is the staff database depicted in Figure 2.3, then changing the value of the position attribute from "PhD Student" to "Postdoc" would be interpreted as an Employee_mod occurrence, but also as an occurrence of Staff_mod.

```
<Staff>
  <Employee
    name="Jan Carlson" position="PhD Student"  ... />
  ...
</Staff>
```

Figure 2.3: Staff database in XML.

2.4 Issues in event pattern detection

Event frameworks that support specification and detection of event patterns come in a variety of types and styles, influenced by the requirements and particulars of different application domains. This section surveys some aspects that distinguish different frameworks, and discusses qualities and properties that are often desired in event pattern detection.

2.4.1 Offline or online detection

In offline event pattern detection, there is a clear separation between the time at which the events occur and the time at which pattern detection takes place. Contrasting this, online detection continuously detects patterns as the primitive events occur, which is needed when the occurrence of a pattern is supposed to trigger some direct response, or when events are used for communication.

If online detection is to be performed for a long time, and in particular if bounds on memory, processing time or detection latency have to be ensured, the detection mechanism must be fast enough to process events in the same rate as they occur, at least when viewed over a longer period of time. As a consequence, storing all past occurrences and searching among them when detecting future patterns, is not a feasible strategy since this eventually becomes too costly. Instead, the detection mechanism typically maintains a state, preferably of bounded size, containing only the information about past occurrences that is required to correctly detect future instances of the pattern.

Harada and Hotta describe a number of problems encountered when developing an online detection system based on a previous version for offline use, such as event occurrences that are reported "out of order" [69].

The main focus of this thesis is online detection, but the proposed event algebra can be used for offline detection as well. In particular, the efficiency of the detection mechanism and the bounded-memory property allow the algebra to be used also for detection of complex patterns on very large event logs. A majority of the languages and frameworks discussed in this thesis address online detection. Examples of offline detection include analysis of medical orders in a hospital computer system [69], computer forensics [1] and extraction of usability information by monitoring events of a graphical user interface [74].

2.4.2 Static or dynamic detection

Online techniques for event pattern detection differ in the level of flexibility they support. In a fully static setting, the patterns are defined before the detection starts, and remain fixed throughout the detection process.

In other frameworks, for example those providing publish/subscribe functionality, the patterns of interest can change dynamically at runtime, which requires more support from the event framework. For example, if all patterns are known at compile time, it is possible to generate specific, optimised code for detecting them. This allows the runtime environment to have little or no knowledge about pattern detection, compared to a dynamic setting where there must be support for processing arbitrary pattern specifications and setting up appropriate mechanisms for detecting them.

Between the static and the fully dynamic approach where new pattern definitions are created at runtime, are frameworks where a number of patterns are defined statically, but the detection of each pattern can be turned on and off dynamically. For example, the language GEM [98] allows rules that are activated or deactivated in response to the detection of a particular pattern. Combined with a delay construct, this makes it possible to specify rule sets that report at most one occurrence of a pattern within a given time interval, as exemplified in Figure 2.4.

rule A**: On pattern** P**, forward** P **and deactivate rule** A
rule B**: On pattern** $P + 5$ sec**, activate rule** A

Figure 2.4: Example of dynamicity in GEM. This rule set ensures that during any five second interval, at most one occurrence of the pattern P is reported to the rest of the system.

2.4.3 Compositionality and event algebras

Composition, i.e., the possibility of combining simple concepts into increasingly more complex concepts, is supported in some form in virtually all languages. In the context of event pattern detection, it is desirable that the composition options make sense from a pattern point of view, and not just with respect to the underlying detection mechanism, since this allows a user to reason about the meaning of a composite pattern

specification without knowing the details of how the detection is implemented. For example, if patterns are defined by ordinary program code, then two definitions can be combined by, for example, sequential composition, conditional jumps, etc., but it is not always straightforward to see what pattern the resulting program detects. So, rather than using the compositions of the underlying detection language, it is advantageous to introduce specialised pattern composition operators to allow specification on a conceptual level. For example, the binary operator *or* combines two patterns P_1 and P_2 into the slightly more complex pattern P_1 *or* P_2. With additional operators, and parentheses to avoid ambiguity, expressions like "P_1 *followed by* ((P_1 *and* P_2) *or* P_3)" can be constructed.

Together, a collection of such pattern operators form an *event algebra*. An *algebra* is a very general mathematical structure, consisting of a domain, a family of operators over this domain, and constants (although constants can formally be represented as operators that take no arguments). In the case of event algebras, the domain consists of possible event occurrence scenarios, and there is typically a constant for each simple primitive event. Then, for a given assignment of an occurrence scenario to each constant, the operator semantics defines the occurrences of compound expressions. For brevity, and to distinguish between formal operators and informal descriptions, operators are usually denoted by single symbols. For example, the pattern "P_1 *followed by* ((P_1 *and* P_2) *or* P_3)" above, is denoted $P_1;((P_1+P_2)\vee P_3)$ in the algebra presented in this thesis. An overview of algebra theory is outside the scope of this thesis, and we refer to Meinke and Tucker [99] or Burris and Sankappanavar [29] for this.

Of course, the concrete detection mechanism implements the pattern operators by means of some low level composition operators, but this is of little concern to the user once they are given their own semantics. When developing the algebra, however, the detection mechanism typically influences the choice of pattern operators, since there are operators that are straightforward to implement in some formalisms but very hard in others. For example, two automata detecting patterns P_1 and P_2, respectively, can be combined into an automaton detecting the composite pattern *"P_1 followed by P_2"* by simply associating the accepting states of the P_1 automaton with the initial state of the P_2 automaton.[1] On the

[1] At least, it can be done with a bit of handwaving. Depending on the desired semantics, it might not be that straightforward, as will be demonstrated later.

other hand, non-occurrence of events is more difficult to express with finite automata. An automaton for the pattern *"P_1 followed by P_2 with no P_3 in between"* is not easily constructed from the automata for the constituent patterns, since an occurrence of P_3 must invalidate any partial detection of the P_2 pattern, and thus affects all states of the P_2 automata.

Detection methods, i.e., the concrete mechanisms that perform the detection, can also be classified with respect to compositionality. With compositional detection, each subpattern can be detected in isolation, independently from whether it is part of a larger pattern or not. For example, to detect the pattern A *followed by* $(B$ *or* $C)$, the subpattern B *or* C is detected separately, and the detected occurrences, together with the primitive occurrences of A, are used as input to the detection of the whole pattern.

2.4.4 Timestamping pattern occurrences

Like primitive occurrences, occurrences of a composite event should be timestamped. The simplest approach is to use the time of detection, and then include the occurrence in the event history just as if it was a primitive occurrence. If this occurrence is later used as the constituent of some other composite event, it can be treated uniformly with the primitive occurrences. However, this simple approach has some drawbacks when it is used in a compositional detection mechanism. For example, consider the pattern A *followed by* $(A$ *or* $B)$ and a single occurrence of A at time t_1. This results in an occurrence of A *or* B, but with a time model of high enough granularity, this occurrence will be given a somewhat later timestamp t_2. Thus, since $t_1 < t_2$, there exists an occurrence of A followed by an occurrence of A *or* B, which results in a detection of the whole pattern, which is probably not the intended meaning.

Alternatively, occurrences of composite events can inherit the timestamp from the occurrences that caused the detection, typically from the latest occurrence [136]. In the example above, this would mean that the A *or* B occurrence is given timestamp t_1. One drawback with this approach is that it introduces simultaneous occurrences also when primitive event occurrences are non-simultaneous.

The difference between occurrence time (i.e., timestamp) and detection time is sometimes assumed to be negligible, in which case the two approaches are essentially the same. However, some frameworks (includ-

ing the event algebras developed by Roncancio [130] and by Galton and Augusto [55]) include operators for composite events that can only be detected later than the time associated with the occurrence. For example, the pattern *"A not followed by B for 2 seconds"* can be said to occur at the time of the A occurrence, but it can only be detected two seconds later. This is not particularly problematic in a framework for offline detection, but an online detection mechanism must be able to deal with occurrences that are detected "out of order", and to delay reporting a detection until the point in time when no future detection of a constituent event can invalidate it.

2.4.5 Pattern parameters

In frameworks where primitive event occurrences are associated with additional information in the form of event parameters (see Section 2.3.2), it is natural to capture this information in the representation of pattern occurrences as well. The most straightforward solution is to associate a pattern occurrence with a collection of parameters from all constituent event occurrences that caused the pattern to occur [136]. Alternatively, the specification language can allow the user to describe, as a part of the pattern specification, how the pattern parameters should be constructed from the parameters of constituent events [48, 105].

Some methods also allow parameters to directly influence the pattern specification [58, 70]. For example, it might be possible to define patterns such as *"two consecutive occurrences of A associated with the same user"* or *"an occurrence of A followed by an occurrence of B with a higher temperature parameter value"*.

2.4.6 Declarative or procedural pattern specification

We also make the distinction between *declarative* and *procedural* techniques for event pattern specification. Procedural methods describe how to achieve the desired result, while a declarative method defines what the desired result is. In event pattern detection, automata based techniques are typical representatives of the procedural approach, explicitly defining the effects of a particular event occurrence, depending on the current state. Declarative techniques include temporal logics, where patterns are described by the relations between the constituent event occurrences.

Generally speaking, declarative specifications ca be easier to manipulate and analyse, but they require some external mechanism that knows how to perform detection of a given pattern, which means that the user has less control over the actual detection process, potentially resulting in less efficient detection.

2.4.7 Single, repeated or overlapping detection

Some applications only require that the first occurrence of the specified pattern is detected, but in other cases such *single detection* techniques are not adequate. A straightforward way to detect multiple pattern occurrences, here termed *repeated detection*, is to simply restart the detection mechanism once the first occurrence is detected and reported. A drawback of this approach, however, is that it only detects non-overlapping pattern instances. Consequently, to determine if a particular instance of a pattern will in fact be detected, we need to look outside the time interval in which it occurs. In fact, we might have to consider the entire sequence of events preceding it.

If, on the other hand, occurrences are detected also when they overlap in time, the detection of a particular pattern instance is independent from other instances of that pattern. We refer to this approach as *overlapping detection*. Methods based on single or repeated detection include ECL [131] and Ode [60], while overlapping detection is used in, e.g., Snoop [35] and chronicle recognition [47]

Figure 2.5 illustrates these three approaches to the detection of the pattern *"A followed by A followed by B"*. If we consider only the event occurrences between times 2 and 5, it is clear that the pattern appear once in this interval (with A occurring at times 2 and 4, followed by a B occurrence at 5). This pattern instance is however not detected with the repeated detection approach, since the detection procedure is restarted after the first detection of the pattern at time 3.

These different detection approaches also have an impact on compositionality. As described above, two automata detecting the first occurrence of patterns P_1 and P_2 respectively, can easily be composed to detect the first occurrence of *"P_1 followed by P_2"*. However, if the two automata are designed to detect *all* occurrences of P_1 and P_2, it is far more complicated to combine them in order to create an automaton that detects all occurrences of *"P_1 followed by P_2"*.

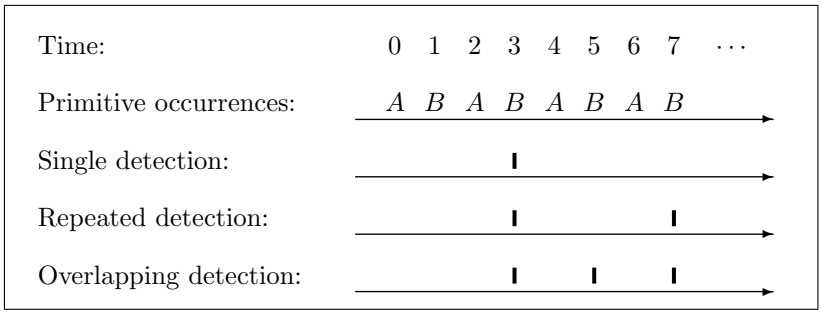

Figure 2.5: Comparison of single, repeated and overlapping detection of the pattern *"A followed by A followed by B"*.

2.4.8 Single point or interval semantics

In most event pattern frameworks, each occurrence of an event pattern is associated with a single time point (typically either the actual time of detection, or the time of the last primitive occurrence that was required to form the pattern). This is termed *single point semantics* (or sometimes *detection semantics* or *termination semantics*). Pattern occurrences are thus considered to be instantaneous, just like primitive occurrences. However, as shown by Galton and Augusto [55], associating a pattern occurrence with a single point in time can result in unintended semantics in some cases of composition.

As an example, consider the sequence operator, with the intuitive interpretation of $A;B$ being *"A followed by B"*. Figure 2.6 illustrates a scenario where we have one occurrence each of A, B and C, occurring in that order. Time flows from left to right in these figures, the top row shows the occurrences of primitive events and the rows below show the detected occurrences of different event patterns. With single point semantics (left column), these occurrences are accepted as an occurrence of the pattern $B;(A;C)$, since there is an occurrence of $A;C$ (associated with the occurrence time of C), preceded by a B occurrence. Consequently, with single point semantics, $B;(A;C)$ has exactly the same meaning as $A;(B;C)$, which does not fit well with the intuitive meaning of sequential composition.

As a solution to this problem, Galton and Augusto propose that occurrences should be associated with intervals rather than single time

points, following the practice of knowledge representation techniques such as Event Calculus [86] and Interval Algebra [9] (see Section 3.1.2). With *interval semantics* (also called *occurrence semantics* or *durative events*), the sequence $A;B$ can be defined to occur only if the intervals of A and B are non-overlapping. In our example, no occurrence of $B;(A;C)$ would be detected, since there is no occurrence of B prior to the interval associated with the occurrence of $A;C$. The result of the interval-based version is depicted in the right column of Figure 2.6.

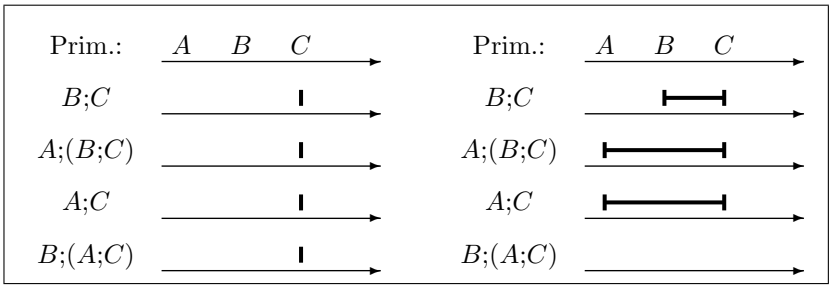

Figure 2.6: Comparison between single point semantics (left) and interval semantics (right).

A problem with interval semantics is that is leads to more complex operator semantics. Two points in time can be related in three different ways ($x < y$, $x = y$ or $x > y$), but, as formalised by Allen [8], there are thirteen possible relations between two intervals. Figure 2.7 illustrates seven of them, and the remaining six are the inverses of these (excluding equality, which is reflexive). For example, consider the semantics of an operator representing the non-occurrence of a pattern within a given interval. Let i be a potential candidate interval and p the only occurrence of the pattern close to this interval. Clearly p *during* i should invalidate the interval, meaning that no non-occurrence should be reported in this case. Other relations, however, are not as easily classified. It is not clear, for example, whether or not i *during* p and p *overlap* i should be considered as valid non-occurrences.

Examples of event frameworks based on intervals include a formalisation of Snoop [3], the work by Roncancio [130] (and the improvement suggested by Gómez and Augusto [64]), Solicitor [100], and ECCO [157], all of which are discussed further in Chapter 3. A different use of inter-

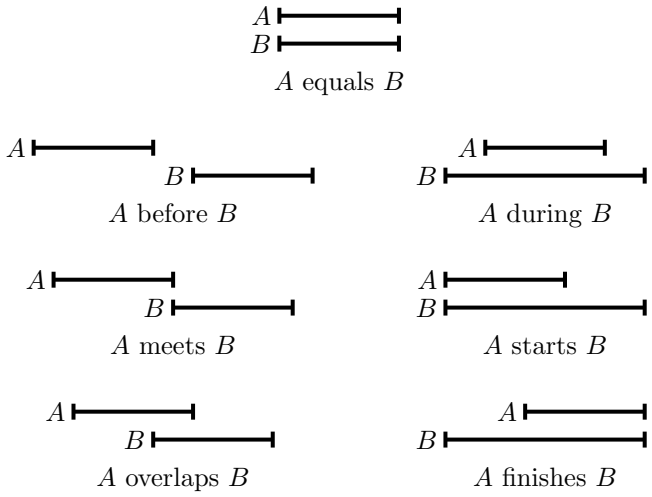

Figure 2.7: Interval relations.

vals in event pattern semantics is presented by Liebig et al. [92], namely as a way to deal with the absence of global clock in distributed systems (see Section 2.3.1). They associate each occurrence (primitive and composite) with an interval indicating the clock uncertainty, meaning that the event occurred at some arbitrary point within the given interval.

2.4.9 Parameter contexts

Declarative approaches to pattern specification and detection are often simple and intuitive to reason about, as discussed in Section 2.4.6. However, they can be difficult to realise efficiently, especially when primitive occurrences carry additional information.

For example, consider the conjunction operator (+), with the intuitive, declarative meaning that $A+B$ occurs when both A and B have occurred, in any order and possibly at different times. In a scenario where we have two occurrence of A, followed by three occurrences of B (as depicted in Figure 2.8), there are in fact six possible combinations of A and B occurrences that match the constraints of the conjunction

operator. In particular, with interval based semantics or when event parameters are used, the six pairs are all different from each other. In some applications, especially those where detection is performed offline, it might be acceptable, or even desirable, to detect all of them as occurrences of $A+B$. However, the memory consumption (each occurrence of A and B must be remembered forever) and the increasing number of simultaneously reported events means that it is unsuitable in many online detection frameworks.

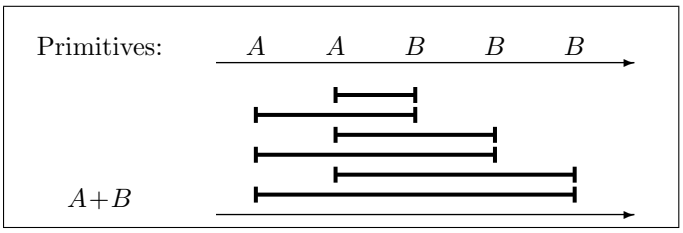

Figure 2.8: Example illustrating the need for parameter contexts. Two occurrence of A followed by three B occurrences result in six potential occurrences of the conjunction $A+B$.

A common approach to deal with this problem is by augmenting the declarative operator semantics with *parameter contexts* (sometimes termed *event contexts*) that specify how occurrences should be selected. Mellin [100] describes the relation between operator semantics and parameter contexts as follows:

> "The event operators aspect addresses the relative constraints between contributing event occurrences, whereas the event contexts aspect addresses the selection of event occurrences from an event stream with respect to event occurrences that are used or invalidated during event monitoring."

For example, the *recent* context in Snoop specifies that only the most recent occurrence of constituent events should be used each time the semantics of the operator is satisfied [35]. For the example in Figure 2.8, this means that only the three pairs containing the second A occurrence is valid in the recent context.

Zimmer and Unland [161] present a formal framework where parameter contexts are based on two orthogonal concepts: An *instance selection*

policy that concerns the selection among constituent occurrences that satisfy the basic operator semantics, and an *instance consumption* policy that defines how a successful detection affects the detection of future occurrences, for example if the same constituent occurrence may be used more than once. A similar approach is used by Zhang and Unger in their event specification language YALES [158]. Other formalisms provide a set of more specialised contexts instead, each of which addresses both selection and consumption. For a concrete example, see Figure 3.3 on page 45 where the four contexts of Snoop are illustrated.

Event contexts can be seen as a pragmatic way to address the efficiency problems related to declarative operators, or, alternatively, as a way to increase the expressive power of a formalism by allowing more detailed specification of which pattern occurrences that are of interest. However, using contexts typically result in a less declarative detection, since selection and consumption depend on previous occurrences. Also, the exact meaning of an operator in a particular context is not always trivial to understand and reason about, especially if there are dependencies, explicit or implicit, between the operator semantics and the contexts.

In addition to the work mentioned above, parameter contexts (or similar constructs) are used in, for example, Solicitor [100], the event algebra developed by Baily and Mikulás [16], ECCO [157], and in the work of Hinze and Voisard [75]. These formalisms are all described further in Chapter 3.

2.4.10 Event correlation

In systems where a single action or situation give rise to a large number of primitive events, there is a need to deal with such event bursts. For example, in a network monitoring system, unplugging a network cable results in a burst of events representing that packets were lost, connection requests failed, acknowledgement timeouts were reached, etc. [95]. While the role of event pattern detection is to identify a particular situation (i.e., when primitive events occur in accordance with the pattern), *event correlation* aims at congregating primitive events that share some characteristics. For example, a network monitoring system might specify that all TIMEOUT events associated with the same physical connection occurring within two seconds should be reported as a single BROKEN-CONNECTION event.

Although event correlation has much in common with event pattern detection, and the separation of the two concepts is somewhat arbitrary, the focus of this thesis is on detecting the occurrences of particular event patterns, rather than on congregating correlated events. Event correlation has been thorougly studied for management of computer and telephone networks [54, 78, 95], but it has also been applied to, e.g., security [66] and power delivery networks [121].

2.5 Embedded and real-time systems

An *embedded system* is a special purpose computer that is built into a larger device. Contrasting general purpose computers, such as PCs, which are designed to support many different types of activities, an embedded system is designated to perform a specific task [148]. Representative examples of embedded systems include handheld devices such as mobile phones and PDAs, cash machines, DVD players, avionic and automotive controllers, but also large stationary systems for example in industrial automation.

Embedded systems are commonly used in *safety-critical* applications, meaning that failure or malfunction may result in serious damage to people, equipment or the environment [144], and in many cases they are supposed to run continuously for years without errors or manual interaction. As a consequence, much effort is spent ensuring that the system behaves according to the specification, both with respect to functional requirements and non-functional aspects such as performance and reliability.

Another characteristic is that embedded systems typically have to function under severe resource limitations compared to general purpose computers, e.g., in terms of available memory, bandwidth and energy [28]. Many embedded systems are subject to *real-time* constraints, meaning that the correctness of the system depends not only on the results it produces, but also on the time at which the results are available.

To cope with issues such as timing and safety, real-time embedded systems are typically structured as a collection of computational activities, called *tasks*, that collaborate to provide the desired system behaviour. Some tasks can execute independently from each other, while others are subject to constraints, e.g., when a result produced by one task is required by another task, their relative order is restricted. When

several tasks are available for execution at the same time, a *scheduler* is responsible for allocating processor time to them according to some predefined scheduling policy. Such policies range from simply following a static schedule that was created offline from the task constraints, to online scheduling policies based on dynamic task properties such as the remaining time to deadline.

One focus of research on real-time systems is to develop suitable scheduling policies for various application types, taking into account factors such as energy consumption, distribution, task dependencies of different kinds, and tradeoffs between worst and average case properties [30]. Another line of research regards techniques to analyse system behaviour under a given scheduling policy, for example to guarantee schedulability, i.e., that there are no circumstances under which a task violates its constraints, or to establish statistically sound estimates of the average response times for system services [83].

2.5.1 Events in embedded real-time systems

In many real-time systems, tasks are executed periodically at predetermined points in time, since this gives a high level of predictability, and because the main functionality of many embedded systems should be carried out continuously, for example controlling a motor based on a desired speed and feedback from sensors. Such *time triggered* execution is not very suitable, however, when the system is supposed to respond to situations that occur rarely, but where a quick response is crucial when they do occur [28]. In a time triggered setting, this means that the task responsible for this activity must be run at a high frequency to ensure that the next activation is always sufficiently close in time, whenever the situation occurs. During time periods when the rare situation does not occur, the periodic task will in fact use very little of the allotted computational resources. Unfortunately, if other tasks must meet their deadlines under all circumstances, they can not depend on this time being available, which leads to poor processor utilisation and possibly a non-schedulable system.

As an alternative to time triggered task activation, *event triggered* activation means that tasks are activated by occurrence of external or internal events, rather than at predefined points in time. When task activation times are unknown, more complex scheduling decisions must be made during runtime. To keep scheduling overhead at a minimum,

many systems use *fixed priority scheduling* (FPS), meaning that each task is assigned a static priority offline, typically based on the relative frequency, urgency and criticality of the task. At runtime, scheduling is reduced to selecting the task with highest priority for execution, from those that are currently ready.

In order to make static timeliness guarantees possible in an event triggered system, the occurrences of events that activate tasks must be bounded in some way, based on knowledge or assumptions about the system environment. For example, *periodic tasks* are activated by events occurring periodically at a given frequency. For other tasks, it might be evident from the nature of the environment that two consecutive occurrences of the activation event are always separated by at least a certain amount of time, called the *minimum interarrival time* (MINT). Such tasks are often referred to as *sporadic*, and tasks with an unbounded arrival rate are termed *aperiodic*.

For perspectives on the difference between time and event triggered systems, and the respective merits of the two approaches, see Kopetz [84], Audsley et al. [15]. The related issue of offline scheduling versus priority based scheduling has been covered by Xu and Parnas [156] and Locke [96]. It should also be mentioned that there are several lines of work that aims at bridging the gap between the two paradigms, or to combine their respective desirable properties. For example, the slot shifting method, developed by Fohler [53], allows sporadic and aperiodic tasks to be handled efficiently in a time triggered setting. Dobrin [46] describe how FPS attributes can be derived from an offline schedule, in such a way that the FPS execution matches the execution of the offline schedule in some respects.

2.6 Component based development

Since one topic of the thesis concerns event detection in the context of a component model for embedded systems, we provide a brief introduction to component based development. For a more complete picture, the reader is referred to existing literature, for example Heineman and Councill [73] or Crnkovic and Larsson [44]. The key principle of component based development (CBD) is to build software systems from existing software units, termed *components*, that are developed separately with reuse and integration in mind. This approach aims to reduce develop-

ment time and the complexity associated with software development, thereby reducing development and maintenance costs.

The central CBD concept *component* can be defined as a software element with well defined interfaces that specify what services the component provides and what services it requires from the environment, i.e., the surrounding components. The exact form of these interface specifications is defined by the *component model*, which also specifies in what ways, and with what results, components can be composed. An important aspect of component composition, which distinguishes it from the mechanical task of combining components that have matching interfaces, is the notion that an assembly of composed components should have properties that can be derived from properties of the constituent parts and the relations between them [143].

Most existing component technologies either target general purpose desktop applications or large distributed systems. Examples of the former include JavaBeans [145] from Sun Microsystems and Microsoft COM [26], while the CORBA Component Model [116], Enterprise JavaBeans [108] and .NET [41] target distributed systems. The component based strategy has been less successful in the area of embedded systems, due to the specific demands of this domain [42, 151]. In particular, with limited resources and strict timeliness requirements, it becomes more difficult to provide predictable composition of components. Ideally, the behaviour of a component should be the same regardless of the environment in which it is deployed, i.e., the other components in the system, but this is not straightforward to achieve for extra functional properties such as timing. Also, the runtime mechanisms needed to replace or reconfigure components dymamically without halting the application, introduce an overhead that is unacceptable for many embedded applications.

Nevertheless, there are component models that specifically target embedded systems, including SaveCCM [6], which is described in Chapter 7, Koala [149] and PECOS [112]. Also, some general purpose component models come in variants that focus on particular aspects that are important to embedded systems, such as Real-Time CORBA [115] and Minimum CORBA [114], addressing issues related to real-time demands and resource limitations, respectively.

2.6.1 Events in component based systems

The event based style of communication is well suited for the component based approach where components are developed without exact knowledge of what other entities they will communicate with in a particular system, and where components may be added and removed dynamically. For example, the JavaBeans specification lists event communication as one of the core features [145]. The mechanisms for subscription and notification follow those in Java [52], with *event listener* interfaces indicating that a class is capable of responding to a particular type of events, and registration methods in event sources that allow event listeners to dynamically subscribe to an event, or remove a subscription. While the subscription is active, the producer and consumer are tightly coupled, since the producer has explicit knowledge of the currently registered subscribers.

The CORBA component model provides a more loosely coupled, distributed publish/subscribe event model, where the distribution of event occurrences to subscribers is done via *event channels* managed by the underlying framework [116].

A recent example of a component based technology that uses events for communication is Microsoft Robotics Studio (MSRS), a development environment for robotic applications [102]. A MSRS application consists of components, called *services*, that communicate and synchronise via event subscription and notification provided by CCR [39], which is the part of the underlying runtime framework responsible for concurrence and coordination.

Chapter 3

Overview of Related Work

This chapter surveys some existing languages and methods for event pattern specification and detection. For a discussion on how these methods relate to the algebra presented in this thesis, see Section 8.2.

3.1 Temporal logic

The aim of of temporal logics and similar formalisms is to formally represent and reason about information with temporal aspects. This is motivated by, for example, work in artificial intelligence such as planning and natural language semantics, but it is also used for example to specify real-time systems [11, 19]. In the context of event pattern detection, these techniques can be used to specify a particular event pattern as a logical formula which is true in that very situation only. Then, the general deduction mechanism of the formalism can be used to determine if, and when, the pattern occurs. In the case of online detection (see Section 2.4.1) this task is simplified if the deduction mechanism supports incremental reasoning when new facts are added (in this case, new facts about primitive event occurrences), to avoid re-evaluating the whole formula every time a new event occurs.

Alternatively, temporal logic can be used as the underlying formalism defining the semantics of a more high-level event pattern specification

technique, as in, for example, the event algebra developed by Baily and Mikulás [16] or in the work of Kiringa [82].

In a temporal logic, the truth value of statements or predicates can vary over time, and thus, so does the truth value of formulas. In addition to the ordinary connectives and quantifiers, most temporal logics also include a number of operators to quantify over time. There are two fundamental ways in which time can be considered. With *linear time*, each moment is considered to have a unique future, while with *branching time* there might be several possible futures [150]. The latter is for example appropriate when reasoning about the possible behaviours of a nondeterministic computer program. This type of branching time is sometimes more precisely termed *right branching time*, distinguishing it from the opposite *left branching time* where each moment may have several possible pasts, which is useful for reasoning "backwards" in time.

Figure 3.1 illustrates these concepts. With linear time, we can make statements such as *"A will occur in the future"* and *"it will always be true that A will occur at some later point in time"*. Branching time allows quantification over the possible future (or past) branches, such as *"it is possible that A never occurs in the future"* and *"up to this point, A has either occurred twice, or not at all"*.

Event pattern detection is in most cases based on a linear model of time, since it is applied to a single concrete sequence of event occurrences, rather than, say, a non-deterministic model of the possible event occurrence scenarios (in which case right-branching time could be useful). It is possible, though, to view a pattern specification as a branching time scenario. For example, Figure 3.2 depicts right and left branching versions of the same pattern. In an offline detection framework, where occurrences can be accessed in any order, this particular pattern is more efficiently detected in the left branching version. The right branching variant must always consider all three branches, but the left branching version can combine them during the detection of the common parts at the end. In an online framework, however, detection would probably follow the right branching version anyway, in order to process occurrences in the order in which they happen.

3.1.1 LTL, CTL* and CTL

Linear temporal logic (LTL) is based on a linear time, and contains temporal quantifiers denoting that a certain formula holds in the next time

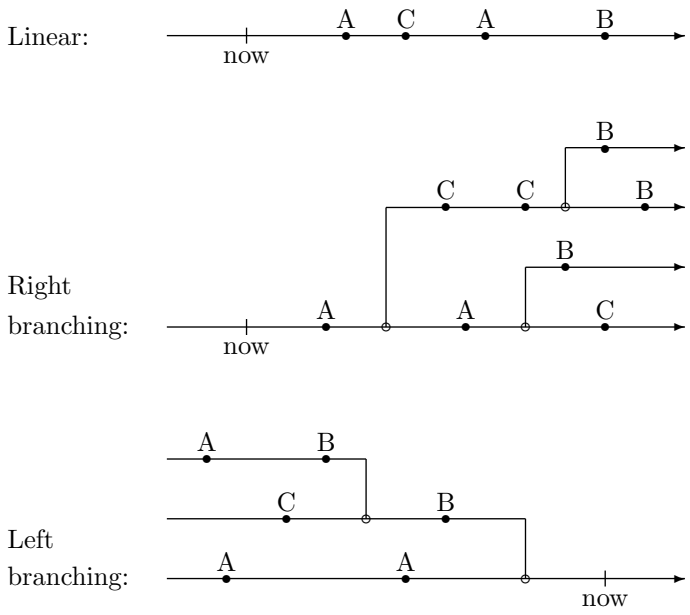

Figure 3.1: Three notions of time: Linear, right branching and left branching.

instant, that it holds always in the future, or eventually in the future, respectively. LTL is a strict subset of *Computation tree logic* (CTL*), which is based on a right branching notion of time. Hence, in addition to the temporal quantifiers of LTL there are two path quantifiers, one representing that a formula holds for all paths, and another denoting the existence of a path for which it holds. In CTL*, path quantifiers and temporal quantifiers can be mixed freely, but there is an important subset, termed CTL, where they are only allowed in pairs consisting of a path quantifiers followed by a temporal quantifier. As a result, verification of a CTL formula can be performed significantly more efficiently. See, for example, the work of Clarke et al. [40] for a formal treatment of this subject.

Sen et al. [137] describe how a deterministic finite automaton can be

Figure 3.2: Right and left branching versions of a pattern.

generated from a given LTL formula. The automaton recognises *good prefix*, meaning that no sequence of subsequent events can make the formula false, and *bad prefixes* for which no combination of future events can satisfy the formula. The method works for any LTL formula, but the size of the automaton monitoring a formula of size m, is $O(2^{2^m})$.

3.1.2 Interval Temporal Logic and Event Calculus

Allen's interval algebra [8] defines thirteen possible relations between temporal intervals, illustrated in Figure 2.7. An algorithm is described by which a network of interval relations can be updated and to some extent checked for inconsistencies when new information is added. Based on this interval algebra, an interval temporal logic is defined that supports reasoning about actions and events [9].

Event calculus [85, 86] is similar in style to the interval temporal logic but avoids the use of non-classical logic. Events are expressed by means of Horn clauses with time as an additional parameter, and the formalism uses *negation as failure*, which means that pattern specifications can be executed as Prolog programs.

These methods are primarily concerned with representing and reasoning about event information, and not so much with the issue of how a particular event pattern can be detected, but in the work of Roncancio [130], the relations of Allen's interval algebra are used as event algebra operators. E.g., "*A starts B*" defines a composite event that occurs when there is an occurrence of B with the same start time, and later end time, as some A occurrence. Gómez and Augusto later suggested an alternative semantics for the operators to avoid some undesired effects [64].

3.1.3 Event specification in FTL and PTL

Sistla and Wolfson [139] present two temporal logics, called *Future time logic* (FTL) and *Past time logic* (PTL), for specifying temporal triggers (i.e., events patterns) in an active database. They argue that some triggers are more naturally expressed with temporal operators that refer to the future (such as *next-time* and *until* in FTL) and others with operators referring to the past (*last-time* and *since* in PTL).

In both FTL and PTL, the standard first-order quantifiers ($\exists$ and $\forall$) are replaced by a so called *freeze quantifier* on the form $[x \leftarrow t]F$ with the meaning that F is evaluated with variable x bound to the current value of t, where "current" refers to the temporal context in which the subformula is being evaluated. For example, the formula

$$[x \leftarrow \mathit{traffic}]\ (\mathit{eventually}\ (\mathit{traffic} > 2 * x))$$

represents that the current value of *traffic* is doubled sometimes in the future, while the formula

$$\mathit{always}\ ([x \leftarrow \mathit{traffic}]\ (\mathit{eventually}\ (\mathit{traffic} > 2 * x)))$$

means that traffic at any point in the future will eventually be doubled.

Sistla and Wolfson also describe an algorithm for detecting patterns defined by expressions in these two languages. Detection of an FTL expression is performed by dynamically maintaining a directed acyclic graph that represents those parts of the pattern that are still to be satisfied. For PTL expressions, on the other hand, the temporal operators are eliminated by transforming them into database queries on past database states.

3.1.4 Event specification in Past FOTL

Chomicki [38] describes how expressions on the contents of a temporal database, in past first-order temporal logic, can be monitored without requiring that the entire database history is stored. Instead, the database is extended with auxiliary relations that contain enough information about past states to correctly detect the given pattern. The auxiliary relations are updated incrementally as time passed, and it can be shown that their size is bounded by the size of the value domains used in the database.

For example, consider the following formula, expressing that the reception of a message is always preceded by the sending of that message:

$$\forall x(\mathit{recieved}(x) \rightarrow (\mathit{sometimes\text{-}in\text{-}the\text{-}past}\ \mathit{sent}(x)))$$

To efficiently monitor this formula, an auxiliary relation is used that specifies all values of x for which "*sometimes-in-the-past sent*(x)" holds, i.e., all messages that has been sent so far. Naturally, the size of this relation is bounded by the domain of the predicate *sent*.

3.1.5 Intrusion detection with EAGLE

The intrusion detection framework presented by Naldurg et al. [111] allows specification of signatures (i.e., event patterns) corresponding to known types of computer system attacks. These signatures are monitored by an online detection algorithm that raises an intrusion alarm upon detection.

Signatures are expressed in EAGLE [17], a language based on a temporal logic with three temporal operators (*next time*, *previous time* and *concatenation*) and a minimal/maximal fixpoint semantics. The semantics of EAGLE is defined for finite traces, i.e., in the general case one must reach the end of the trace before it can be determined if a given formula is true or false. To use EAGLE for online detection in systems that in theory runs forever, an attack pattern must be specified as a formula which turns true when the pattern occurs, rather than as a general statement over finite traces.

3.2 Automata based detection

Many event pattern specification frameworks are based on some variant of finite state automata, as they provide a formal, well understood and easily implemented foundation. In particular, they can easily be implemented with limited and predictable resource usage.

A detection automaton consists of a finite number of states, including one initial state and a number of accepting states. States are connected by transitions labelled with the primitive events of the system. Starting in the initial state, the automaton follows a transition leading out of the current state at the occurrence of the primitive event labelling that transition. If the transition leads to an accepting state, a detection is

signalled, otherwise the procedure is repeated. After a detection, some frameworks reset the automaton to the initial state, resulting in *repeated detection* (see Section 2.4.7). Other frameworks let the automaton remain in the accepting state once the detection is signalled, and treat subsequent occurrences in the same way as before the detection, for example to achieve a more declarative detection.

The basic notion of automata can be extended in a number of ways. Allowing multiple outgoing transitions with the same label, or empty transitions that can be followed at any time, introduces *non-determinism*, which does not increase the expressive power but can significantly reduce the size of the automata. Similarly, allowing variables that can be assigned and tested on the transitions, can be seen as shorthand for a larger automaton without variables, provided that the variables only take values from some finite domain. For patterns where the occurrences times are significant, and not only the relative order of occurrences, the automata must be augmented with some concept of time. For example, the timed automata formalism includes real-valued clocks that can be tested and reset [10].

Regular expressions have the same expressive power as finite automata [77], but provide a more readable and concise notation. In the context of event pattern detection, the ordinary regular expression operators (e.g., concatenation, alternation and iteration) are sometimes supplemented by additional operators to simplify the specification of common pattern types. These can either be derived from the basic operators or related to particular extensions such as timing or event parameters.

Motaki and Zaniolo [109] point at three main drawbacks with automata based methods for event pattern specification:

1. Parameterised events. If primitive event occurrences carry additional information, as discussed in Section 2.3.2, the standard finite automata concept must be extended to allow this information to be taken into consideration. Although this can be done, it impairs the simplicity of the original formalism.

2. Simultaneous occurrences. Automata based methods typically work on a conceptual sequence of primitive event occurrence, i.e., assuming that the occurrences of primitive events are non-simultaneous. Simultaneous occurrences can be supported by labelling transitions with combinations of events, but this result in significantly more complex automata.

3. Exponential blow-up. Constructs that require that the constituent patterns are detected concurrently, such as conjunction and non-occurrence of complex patterns, typically result in very large automata compared to the size of the automata detecting the constituent parts.

Simultaneous occurrences can also be supported by introducing an explicit time event, meaning that events occurring between two consecutive time events should be considered as simultaneous. For this approach to be reasonable, it should be shown that simultaneous events can represented in arbitrary order without changing the overall outcome of the automaton.

As a consequence of the exponential blow-up caused by concurrent detection, automata do not provide good support for specifying non-occurrence of patterns, such as in the pattern "P_1 *unless* P_2", meaning that an occurrence of the pattern P_2 should invalidate any partially detected P_1 pattern. Non-occurrence of single, primitive events is easily expressed and typically provided as a negation operator in a regular expression language, but non-occurrence of complex patterns is less straightforward. Detecting pattern instances that partially overlap in time, termed *overlapping detection* in Section 2.4.7, also typically require complex automata.

In addition to the work described below, approaches based on automata and regular expressions include the active database system Ode (described in Section 3.3.1), the general event pattern specification language SEL [160], the event language developed by Hayton et al. [72], and the BMSL language for specification of event based security properties, developed by Bowen et al. [25].

3.2.1 Composite event detection automata

Pietzuch et al. [122, 123] describe a general framework for event pattern detection that is based on finite automata with some extensions. The framework targets distributed systems, where it is not always possible to determine the exact ordering of all event occurrences, especially between occurrences originating from different nodes. This is reflected by one of the extensions to the ordinary automata constructs, namely that there are two types of transitions. *Strong* transitions can only be traversed if it can be established that the associated event definitely occurred later than the previously consumed occurrences. *Weak* transitions only

require that the occurrences are correctly ordered with respect to their latest possible occurrence times (considering clock drift bounds, etc.).

To allow patterns with temporal constraints to be specified, the automata can contain *generative* states that, when reached, publish a new event occurrence after a specified amount of time. This event can be used on subsequent transitions in the automaton, for example to discard a partial detection that does not meet the temporal constraints of the desired pattern, or to mark the end of a non-occurrence interval. There is also a construct that detects two patterns in parallel, and succeeds once both have been detected. This is accomplished by multiple, interacting automata running in parallel, but since the semantics of the extended automata framework is only defined informally, the exact semantics of this construct is unclear.

3.2.2 ECL and PAR

ECL is a language for event pattern specification developed by Sánchez et al. [131]. They also define the equally expressive sublanguage PAR to simplify analysis, in particular on pattern specification equivalence [132]. The PAR language resembles regular expressions in style, and any pattern specified in PAR can be detected with bounded resources. A central result is that the opposite holds as well, i.e., that every event pattern that can be detected by a finite state automaton, and thus by any reasonable[1] finite memory formalism, can be expressed in PAR [133]. The assumptions, under which this property is proved, are that the input is a sequence of elements from a finite set of events, i.e., no timestamps or additional information is associated with the occurrences, and no simultaneous occurrences are allowed.

In addition to operators that correspond to the standard operators of regular languages (i.e., concatenation, alternation and iteration), PAR contains an explicit output operator which, for example, allows a successful detection of a particular subpattern to be signalled, an not only the detection of the full pattern. There is also a binary preemption operator for specifying non-occurrences. As an example, the pattern "*try* P_1 *unless* P_2" results in parallel detection of the two subpatterns P_1 and P_2, and succeeds if P_1 is detected unless P_2 is detected first, in

[1] By *reasonable* they mean methods that provide immediate reactions and satisfy determinism and causality.

which case the whole detection fails. The PAR language normally assumes single detection, but repeated detection can also be achieved since the language includes a repetition operator which restarts the detection procedure when an occurrence is detected.

3.3 Active databases

Active databases, unlike ordinary, passive databases, have the ability to react automatically to situations that arise within or outside the database [120]. These reactions are typically specified by ECA rules stating that when a certain event occurs, the given action should be performed if the condition is satisfied. The event part of an ECA rule can include an event algebra to allow the database to react to complex event patterns.

The initial research in the area of active databases was to a large extent empirical in nature, and resulted in a number of approaches that share some characteristics, but with different languages and underlying execution models, often lacking formal semantics [119]. For example, the detection mechanisms in Snoop, SAMOS and Ode are based on different strategies, which introduce some subtle differences in the way patterns are defined although their respective event algebras look quite similar. Later research has been aimed at providing formal foundations for these and other methods, and establishing common underlying theory that unifies the different approaches. For example, Zimmer and Unland [161] present a formal event context framework (see Section 2.4.9) in which the event algebras of Ode, SAMOS, Snoop and a few other systems are compared. They also highlight a number of ambiguities and inconsistencies of the various approaches.

The event algebra presented by Baily and Mikulás [16] follows the Zimmer and Unland framework. It is defined formally in temporal logic and includes four event contexts. They identify a class of composite events for which testing for event equivalence is decidable, and show that testing for implication is undecidable. I.e., in general it is not possible to check for two composite events whether an occurrence of the first always implies that the other occurs as well.

Motakis and Zaniolo have developed the *Event pattern language* (EPL) with a semantics based on Datalog rules [109]. Datalog is a subset of Prolog with restrictions that ensure more efficient querying, such as not allowing complex terms as arguments of predicates. They also show how

event pattern definitions from Ode, SAMOS and Snoop can be translated into equivalent, or in some case similar, EPL expressions, thereby providing a basis for comparing the three languages.

An overview of other research initiatives to formalise different aspects of active databases is given by Paton et al. [119].

3.3.1 Ode/COMPOSE

The event expression language COMPOSE [59] extends the active object database Ode [60] with support for composite events. The language consists of four basic operators and sixteen additional operators, most of which are derived from the basic ones, including for example disjunction, negation and simultaneous conjunction. There are also counting operators that can refer explicitly to the nth occurrence of an event, or to every nth occurrence. The formalism is based on a global, totally ordered set of primitive event occurrences, implying that primitive events can not occur simultaneously.

The operators of COMPOSE has the same expressive power as regular expressions [61], which allows the detection mechanism to be implemented by finite state automata. To support event parameters, and composite events that occur only under given restrictions on the parameters of the constituent events, the automata mechanism is extended with data structures that store the parameters of past event occurrences.

3.3.2 SAMOS

In the active object-oriented database SAMOS [57, 58], event detection is defined and implemented using Petri nets. The event algebra consists of operators for disjunction, conjunction and sequence. There are also three constructs that take an interval specification as an additional argument, namely a counting construct that is signalled every time n occurrences of an event has happened within the interval, an operator that only signals the first occurrence within the interval, and a negation that occurs at the end of the interval if the interval contained no occurrence of the given event. For these constructs, interval start and end points can be specified either as absolute time points or relative to the occurrences of some event.

SAMOS events can have parameters identifying, e.g., which transaction and user a particular occurrences is associated with, and parameter

equivalence restrictions can be included in the pattern specifications. For example, a sequence $A;B$ can be decorated with a restriction that only cases when the A occurrence and the following B occurrence are associated with the same user, should be considered. To support this, the semantics is based on coloured Petri nets [80], an extension of ordinary Petri nets in which each token has a value that can be tested and manipulated by the transitions.

3.3.3 Snoop

Snoop [34, 35] is an event specification language developed to extend a passive database system with active functionality, but it can be used as a standalone event detection system as well. The operators of Snoop are disjunction, conjunction, sequence, negation, aperiodic and periodic. Conjunction also comes in a more general variant, representing that n out of m given events have occurred (with $n = m = 2$, this is equivalent to the binary conjunction operator).

The aperiodic and periodic constructs take three arguments, two of which define the start and end of an interval during which the composite event is active. The aperiodic event $A(E_1, E_2, E_3)$ occurs each time E_2 occurs inside an interval started by an E_1 occurrence and ended by an occurrence of E_3, and the periodic event $P(E_1, T, E_3)$ occurs periodically with period T during the interval from E_1 to E_3. Both of these constructs also come in cumulative versions (A^* and P^*, respectively) in which no occurrences are signalled inside the interval. Instead, the corresponding values are accumulated and associated with a single occurrence at the end of the interval.

Snoop also defines four event contexts. We describe them informally by their effect on the sequence $A;B$, rather than giving the general definition. In the *recent* context, an occurrence of B is only combined with the most recent occurrence of A. In *chronicle*, the first occurrence of A is combined with the first occurrence of B, the second A with the second B, etc. The *continuous* context means that a B occurrence is combined with all earlier A occurrences that have not yet been combined with a B occurrence. Finally, in *cumulative*, an occurrence of B results in at most one occurrence of $A;B$, but the occurrence carries a collection of values gathered from all earlier occurrences of A (but these values are not included again in subsequent occurrences). An example illustrating the four contexts is given in Figure 3.3.

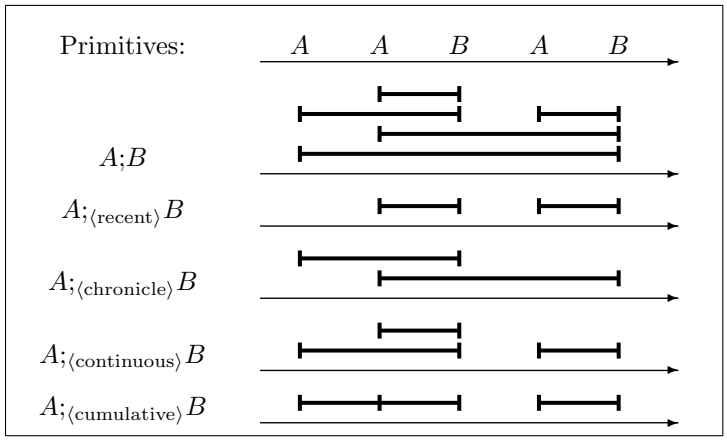

Figure 3.3: Detection of the sequence $A;B$ in various Snoop event contexts.

The detection mechanism includes a specific procedural implementation for each operator/context combination, and the detection of a particular event pattern is structured according to the event expression tree. Occurrences of primitive events are inserted at the leaves, causing the execution of the implementation associated with the operator in the parent node. As more complex events are detected, event occurrences are propagated upwards in the tree.

Early work on Snoop defined operator semantics formally, but the event contexts were only described informally. Later, Chakravarthy and Yang formalised the recent, chronicle and continuous contexts [36]. Also, Adaikkalavan and Chakravarthy [3, 4] developed an interval based semantics (see Section 2.4.8) for some of the Snoop operators in the recent, cumulative and chronicle contexts.

3.4 Embedded and real-time systems

Event detection frameworks that target the embedded systems domain have to address resource issues in some way. For applications with hard real time constraints, bounds on memory and processing time must be determined statically. Some formalisms, such as finite automata, are

inherently resource bounded, but other methods require careful design and analysis.

The work of Hansson and Berndtsson [21, 68] and Sivasankaran et al. [140] on active real-time databases is clearly relevant to the work presented in this thesis. They address how active functionality in the form of ECA rules can be handled in a database that is subject to real-time constraints, which includes, e.g., allocation of detection activities to tasks, priority assignment of events and transactions, and appropriate coupling modes for ECA rules describing how information is transfered between the event, condition and action parts of a rule.

3.4.1 Specifying event patterns in RTL

Mok et al. present a framework based on real time logic (RTL), which is a first order logic with a dedicated predicate encoding event occurrences. The framework has been used in the context of network management [95] and in an electronic brokerage application [105]. It has also been suggested as the basis for composite event specification in active databases [94].

Composite events are expressed as RTL timing constraints and handled by general RTL monitoring techniques [105]. In addition to composing whole events as in *"an occurrence of A or an occurrence of B"*, it is possible to refer to individual event occurrences, which allows patterns such as *"the second occurrence of A followed by an occurrence of B"*. Like the temporal logics discussed in Section 3.1, RTL can be used to specify general properties, and not just event patterns. Consequently, as well as detecting that a specification is satisfied (e.g., that the specified event pattern has occurred), violations must also be recognised. In particular, RTL provides early detection of temporal constraint violation, meaning that the violation of a specification such as *"an occurrence of A should be followed by a B within 2 seconds"*, is reported immediately at the end of that time interval, and not when B finally occurs.

For a particular class of specifications (informally, conjunctions of simple constraints on pairs of event occurrences), violation and satisfaction detection algorithms are presented that run in $O(n)$ time at each check point [103]. The number of event occurrences that must be stored by the detection algorithm is bounded, as is the number of simultaneously active timeouts, and these bounds can be derived from the constraints at compile time [103].

3.4.2 Solicitor

The event specification language Solicitor [100, 101], developed by Mellin and Andler, also targets real-time systems in particular. The operators and parameter contexts resemble those in Snoop, but Solicitor is based on a formal schema where event operators and contexts are defined as fully orthogonal concepts. This separation makes it easier to reason about the algebra, compared to frameworks where each combination of operator and context requires an individual formal definition.

The memory requirement of the detection mechanism is bounded, under the assumptions that a minimum interarrival time is known for each primitive event in the system, and each composite event is associated with an expiration time that restricts the time between the earliest and latest occurrence that may form an occurrence of the composite event.

3.5 Additional work on event notification

When event pattern detection is viewed in a broader perspective, for example as one aspect of the event support provided by a middleware, or in a general purpose notification framework, there are many factors to consider in addition to the specification and detection of patterns. When ECA rules are used, the specification and evaluation of the condition and action parts of rules, and how information is transfered between the three parts, must be addressed. There are also issues concerning rule management, such as dynamic introduction of new rules and scheduling of rules that may trigger at the same time. In distributed systems, there are additional problems related to timing. In particular, primitive event occurrences originating from different nodes can not always be totally ordered as a result of clock synchronisation inaccuracy. Also, the time it takes to propagate an event occurrence to the node at which pattern detection is performed must be take into account.

READY is an event notification service, developed by Gruber et al. [65]. It targets distributed, heterogeneous systems based on the publish/subscribe paradigm, and contains a simple event algebra for registering composite events. The ECCO framework, developed by Yoneki, addresses asynchronous group communication over wireless ad hoc networks [157]. Subscription to composite events is handled by a mechanism based on our algebra and on the work of Pietzuch et al. [123]. The work by Hinze and Voisard [75, 76] on event notification services includes an

event algebra for specifying composite events. The algebra is parameterised with respect to policies for event instance selection and consumption, but the two concepts are not treated independently as suggested in the work of Zimmer and Unland [161].

3.5.1 GEM

GEM, by Mansouri-Samani and Sloman [98], is a declarative event monitoring language for distributed systems, where special consideration is given to problems related to the delay introduced when patterns consist of events detected on different nodes. The event pattern specification part of GEM is interval based, and a noticeable detail is that the language includes constructs that explicitly refer to the start and end time of occurrences, to be used in condition expressions. Thus, it is possible to express both patterns such as *"A and B occurs in any order within 2 seconds"*, but also *"an A occurrence and a B occurrence with the same start time"*. The action part of a GEM rule can publish new event occurrences in response to a complete detection, but they can also dynamically activate or deactivate certain rules, as exemplified in Section 2.4.2.

3.5.2 Chronicle recognition

Chronicle recognition [47, 48] stems from work in the domain of artificial intelligence and knowledge representation, and addresses the detection of a certain type of event patterns called *chronicles*. A chronicle is specified by a set of events that must occur and a set of temporal constraints over their respective occurrence times. Furthermore, it can contain assertions stating that a certain property must hold during an interval with start and end times given by two of the constituent event occurrences, and finally a set of forbidden events that invalidates the pattern should they occur within the specified interval. Figure 3.4 gives an example of a chronicle specification, in a somewhat simplified syntax. The full syntax also includes specifications of the desired reaction and occurrence criteria for the primitive events.

The chronicle descriptions are translated into graphs of temporal constraints, in a preprocessing phase. The online detection mechanism manages a collection of partially satisfied chronicles, and their respective temporal constraints graph. When a primitive event occurs, it is included in partial chronicles that still wait for that event, if this is al-

chronicle E {
    A **occurs at time** a_1
    A **occurs at time** a_2
    B **occurs at time** b
    $2 < a_2 - a_1 \leq 6$
    $a_1 < b < a_2$
    no C **occurrence in** $[a_1, b]$
}

Figure 3.4: Chronicle specification example (simplified syntax).

lowed by the temporal constraint graph. Partial chronicles that contain a currently open interval of non-occurrence of this event are invalidated, i.e., removed from the collection. Invalidation can also be triggered when a certain time point is reached, if it is the case that no future occurrence can satisfy the temporal constraints.

Before an occurrence is included in a partial chronicle, the chronicle is duplicated, and the occurrence is only included in one of the copies. Thus, the collection will investigate the possibility that the current occurrence eventually will contribute to the detection of the whole chronicle, but also, in parallel, the cases where some other occurrence is used in its place. As a result, chronicle recognition can detect pattern instances that overlap in time (see Section 2.4.7).

Chapter 4

The Event Algebra

This chapter presents the syntax and semantics of the proposed event algebra, and establishes algebraic laws and other significant properties. The most important consideration when developing the algebra has been that the formal meaning of a pattern specification should differ as little as possible from the informal, expected behaviour, without preventing detection with bounded memory. The semantics is interval based to ensure that operators obey desired algebraic laws, and resource efficiency is addressed by means of a declarative restriction policy, similar in style to a parameter context.

The algebra is primarily intended for online detection, although the bounded memory property makes it an interesting candidate also for offline detection over very large event sequences. It provides overlapping detection, meaning that pattern instances are detected also when they are temporally overlapping. Primitive events can have parameters, i.e., additional information associated with the individual occurrences, and simultaneously occurring primitive events is supported.

4.1 Preliminaries and syntax

We assume a discrete time model, but the declarative semantics of the algebra could be used with a dense time model as well, under restrictions that prevent primitive events that occur infinitely many times in a finite time interval.

Definition 4.1. *The temporal domain $\mathcal{T}$ is the set of natural numbers.*

Definition 4.2. *Let $\mathcal{P}$ be a finite set of identifiers that represent the primitive event types that are available to the system.*

Definition 4.3. *If $A \in \mathcal{P}$, then A is a primitive event expression. If A and B are event expressions, and $\tau \in \mathcal{T}$, then $A \vee B$, $A+B$, $A-B$, $A;B$ and A_τ are composite event expressions.*

Informally, a *disjunction* $A \vee B$ represents that either of A and B occurs. A *conjunction* means that both events have occurred, in any order and possibly at different times, and is denoted $A+B$. The *negation*, denoted $A-B$, occurs when there is an occurrence of A during which there is no occurrence of B. A *sequence* $A;B$ is an occurrence of A followed by an occurrence of B. Finally, there is a *temporal restriction* A_τ which occurs when there is an occurrence of A shorter than τ time units.

Example 4.1. Continuing the running example from the introduction, we consider a system with a button B, a pressure alarm P and a temperature alarm T, where some action should be performed every time the button is pressed twice within two seconds, unless either of the alarms occurs in between. For this system we have $\mathcal{P} = \{\mathsf{B}, \mathsf{P}, \mathsf{T}\}$, and the described situation can be defined by the expression $(\mathsf{B};\mathsf{B})_{2\ \mathrm{sec}} - (\mathsf{P} \vee \mathsf{T})$ in the algebra. ◇

Occurrences are represented by *event instances*. Since the information associated with an occurrence varies between different applications, we define an underlying abstract framework rather than providing a single concrete representation. Primitive event occurrences are instantaneous and atomic, but composite occurrences are associated with time intervals rather than single time points. As discussed in Section 2.4.8, this is required in order to achieve some of the desired algebraic properties.

The interval of an event instance e is captured by the functions start(e) and end(e), where end denote the time of occurrence, and the full interval from start(e) to end(e) represents the smallest interval containing everything that caused e to occur. The framework also contains a constructor operator $\oplus$ by which composite instances can be constructed. E.g., each instance of $A;B$ will be constructed from one instance of A and one instance of B.

Definition 4.4. *An instance framework consists of:*

- *a domain D of event instances;*
- *a commutative and associative function $\oplus : D \times D \to D$;*
- *a function* $\text{start} : D \to \mathcal{T}$ *with* $\text{start}(e \oplus e') = \min(\text{start}(e), \text{start}(e'))$ *for any $e, e' \in D$; and*
- *a function* $\text{end} : D \to \mathcal{T}$ *with* $\text{end}(e \oplus e') = \max(\text{end}(e), \text{end}(e'))$ *for any $e, e' \in D$.*

Example 4.2. For systems where no additional information is associated with event occurrences, event instances can simply be represented as start and end time tuples. This would correspond to an instance framework where:

- $D = \{\langle s, e\rangle \mid s, e \in \mathcal{T}\}$;
- $\langle s, e\rangle \oplus \langle s', e'\rangle = \langle \min(s, s'), \max(e, e')\rangle$;
- $\text{start}(\langle s, e\rangle) = s$; and
- $\text{end}(\langle s, e\rangle) = e$.

◇

Example 4.3. In some applications it is useful to tag event occurrence with additional information, e.g., to be used in the responding action. For $A \in \mathcal{P}$, we let $\text{dom}(A)$ denote the domain of values associated with occurrences of A, and define the following instance framework:

- D is the powerset of $\{\langle p, v, \tau\rangle \mid p \in \mathcal{P},\ v \in \text{dom}(p),\ \tau \in \mathcal{T}\}$;
- $e \oplus e' = e \cup e'$;
- $\text{start}(e) = \min(\{\tau \mid \langle p, v, \tau\rangle \in e\})$; and
- $\text{end}(e) = \max(\{\tau \mid \langle p, v, \tau\rangle \in e\})$.

In our example system, the temperature alarm occurrences might carry the measured temperature value, while the pressure alarm is less sensitive and only indicate whether the pressure is too high or too low. The button instances carry no additional information, which is represented by a dummy value $\perp$ in the framework. This corresponds

to $\mathrm{dom}(\mathsf{T}) = R$, $\mathrm{dom}(\mathsf{P}) = \{\mathrm{high}, \mathrm{low}\}$ and $\mathrm{dom}(\mathsf{B}) = \{\bot\}$. Then $\{\langle \mathsf{T}, 38.5, 6\rangle\}$, $\{\langle \mathsf{P}, \mathrm{low}, 4\rangle\}$ and $\{\langle \mathsf{P}, \mathrm{low}, 4\rangle,\ \langle \mathsf{B}, \bot, 6\rangle\}$ are examples of event instances in this framework. ◇

For some applications, it might be more convenient to use a construction operator that does not satisfy the commutativity and associativity requirements. Most results presented in this thesis hold for such frameworks as well (the exceptions are discussed in Section 4.3, on page 60).

Together, all occurrences of a certain event (primitive or composite) form an *event stream*. We require that primitive event occurrences are instantaneous, and that the occurrences of a primitive event are separated in time, although several primitive events can occur simultaneously. An *interpretation* represents a particular scenario, as it captures one of the possible ways in which the primitive events can occur.

Definition 4.5. *An event stream is a set of event instances. A primitive event stream is an event stream S for which the following holds:*

1. $\forall e\ (e \in S \Rightarrow \mathrm{start}(e) = \mathrm{end}(e))$
2. $\forall e\ \forall e'\ ((e \in S \wedge e' \in S \wedge \mathrm{end}(e) = \mathrm{end}(e')) \ \Rightarrow\ e = e')$

Definition 4.6. *An interpretation is a function $\mathcal{I}$ mapping each identifier in $\mathcal{P}$ to a primitive event stream.*

Example 4.4. Using the framework from Example 4.2, the following interpretation corresponds to a particular scenario with two occurrences of T and one occurrence each of P and B:

$$\mathcal{I}(\mathsf{B}) = \{\langle 6, 6\rangle\} \qquad \mathcal{I}(\mathsf{P}) = \{\langle 4, 4\rangle\} \qquad \mathcal{I}(\mathsf{T}) = \{\langle 1, 1\rangle,\ \langle 6, 6\rangle\}$$

In the more detailed framework of Example 4.3, the same scenario might be represented as

$$\begin{aligned} \mathcal{I}(\mathsf{B}) &= \{\{\langle \mathsf{B}, \bot, 6\rangle\}\} \\ \mathcal{I}(\mathsf{P}) &= \{\{\langle \mathsf{P}, \mathrm{low}, 4\rangle\}\} \\ \mathcal{I}(\mathsf{T}) &= \{\{\langle \mathsf{T}, 38.2, 1\rangle\},\ \{\langle \mathsf{T}, 38.5, 6\rangle\}\} \end{aligned}$$

◇

The naming convention is to use S, T and U for event streams, and A, B and C for event expressions. Lower case letters are used for event instances. In general, s belongs to the event stream S, and a to an event stream defined by A, etc.

4.2 Semantics

The following functions over event streams form the core of the algebra semantics, as they define the basic meaning of the five operators.

Definition 4.7. *For event streams S and T, and for $\tau \in \mathcal{T}$, define:*

$$\begin{aligned}
\mathrm{dis}(S,T) &= S \cup T \\
\mathrm{con}(S,T) &= \{s \oplus t \mid s \in S \wedge t \in T\} \\
\mathrm{neg}(S,T) &= \{s \mid s \in S \wedge \neg\exists t(t \in T \wedge \mathrm{start}(s) \leq \mathrm{start}(t) \wedge \\
&\qquad\qquad \mathrm{end}(t) \leq \mathrm{end}(s))\} \\
\mathrm{seq}(S,T) &= \{s \oplus t \mid s \in S \wedge t \in T \wedge \mathrm{end}(s) < \mathrm{start}(t)\} \\
\mathrm{tim}(S,\tau) &= \{s \mid s \in S \wedge \mathrm{end}(s) - \mathrm{start}(s) \leq \tau\}
\end{aligned}$$

The semantics of the algebra is defined by recursively applying the corresponding function for each operator in the expression.

Definition 4.8. *The meaning of an event expression for a given interpretation $\mathcal{I}$ is defined as follows:*

$$\begin{aligned}
[\![A]\!]^{\mathcal{I}} &= \mathcal{I}(A) \text{ if } A \in \mathcal{P} & [\![A-B]\!]^{\mathcal{I}} &= \mathrm{neg}([\![A]\!]^{\mathcal{I}}, [\![B]\!]^{\mathcal{I}}) \\
[\![A \vee B]\!]^{\mathcal{I}} &= \mathrm{dis}([\![A]\!]^{\mathcal{I}}, [\![B]\!]^{\mathcal{I}}) & [\![A;B]\!]^{\mathcal{I}} &= \mathrm{seq}([\![A]\!]^{\mathcal{I}}, [\![B]\!]^{\mathcal{I}}) \\
[\![A+B]\!]^{\mathcal{I}} &= \mathrm{con}([\![A]\!]^{\mathcal{I}}, [\![B]\!]^{\mathcal{I}}) & [\![A_\tau]\!]^{\mathcal{I}} &= \mathrm{tim}([\![A]\!]^{\mathcal{I}}, \tau)
\end{aligned}$$

To simplify the presentation, we will use the notation $[\![A]\!]$ instead of $[\![A]\!]^{\mathcal{I}}$ when the choice of $\mathcal{I}$ is obvious or arbitrary.

Example 4.5. Let $\mathcal{I}$ be the interpretation defined in Example 4.4. This scenario gives the following result, for the simple framework and for the framework with values, from Examples 4.2 and 4.3, respectively:

		Simple framework	Framework with values
$[\![\mathsf{B} \vee \mathsf{P}]\!]^{\mathcal{I}}$	=	$\{\langle 4,4\rangle,\ \langle 6,6\rangle\}$	$\{\{\langle \mathsf{P}, \mathrm{low}, 4\rangle\},\ \{\langle \mathsf{B}, \bot, 6\rangle\}\}$
$[\![\mathsf{P}+\mathsf{T}]\!]^{\mathcal{I}}$	=	$\{\langle 1,4\rangle,\ \langle 4,6\rangle\}$	$\{\{\langle \mathsf{P}, \mathrm{low}, 4\rangle, \langle \mathsf{T}, 38.2, 1\rangle\},$ $\{\langle \mathsf{P}, \mathrm{low}, 4\rangle, \langle \mathsf{T}, 38.5, 6\rangle\}\}$
$[\![\mathsf{T};\mathsf{B}]\!]^{\mathcal{I}}$	=	$\{\langle 1,6\rangle\}$	$\{\{\langle \mathsf{T}, 38.2, 1\rangle, \langle \mathsf{B}, \bot, 6\rangle\}\}$
$[\![(\mathsf{P}+\mathsf{T})-\mathsf{B}]\!]^{\mathcal{I}}$	=	$\{\langle 1,4\rangle\}$	$\{\{\langle \mathsf{P}, \mathrm{low}, 4\rangle, \langle \mathsf{T}, 38.2, 1\rangle\}\}$
$[\![(\mathsf{P}+\mathsf{T})_2]\!]^{\mathcal{I}}$	=	$\{\langle 4,6\rangle\}$	$\{\{\langle \mathsf{P}, \mathrm{low}, 4\rangle, \langle \mathsf{T}, 38.5, 6\rangle\}\}$

Figure 4.1 presents this scenario graphically. ◇

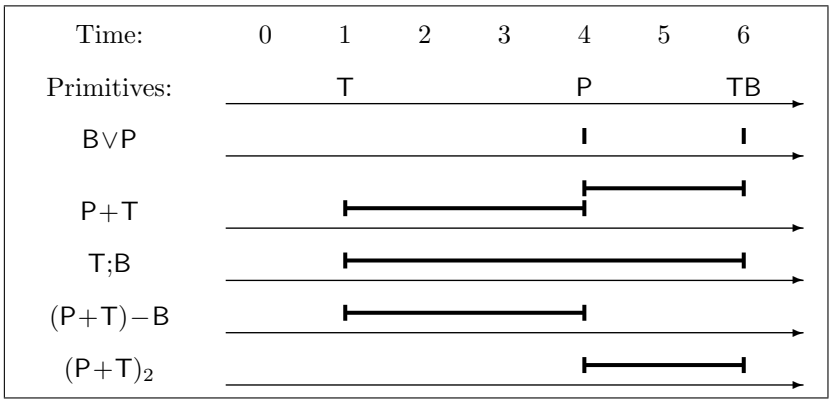

Figure 4.1: Graphical representation of Example 4.5.

These definitions result in an algebra with simple semantics and intuitive algebraic properties, but which can not be implemented efficiently. In particular, the sequence and conjunction operators result in many simultaneous occurrences, and detecting all of them correctly requires that all occurrences of some constituent events are stored throughout the system lifetime.

Example 4.6. Figure 4.2 shows the detection of the expression T+P. Whenever there is an occurrence of T it should be combined with all previous occurrences of P to create instances of T+P, and vice versa. Thus, every occurrence of T and P must be stored for future use. ◇

To deal with resource limitations, we introduce a formal restriction policy that defines a subset of instances that must be detected. The basic idea is to ignore simultaneous occurrences, while at the same time retaining the desired properties of the semantics.

The restriction policy is defined as a binary relation *res* over event streams, where $res(S, S')$ means that S' is a valid restriction of S. Alternatively, it can be seen as a non-deterministic restriction function, or a family of acceptable restriction functions. Rather than computing $[\![A]\!]$ for a given event expression A, an implementation of the algebra should compute an event stream S' for which $res([\![A]\!], S')$ holds.

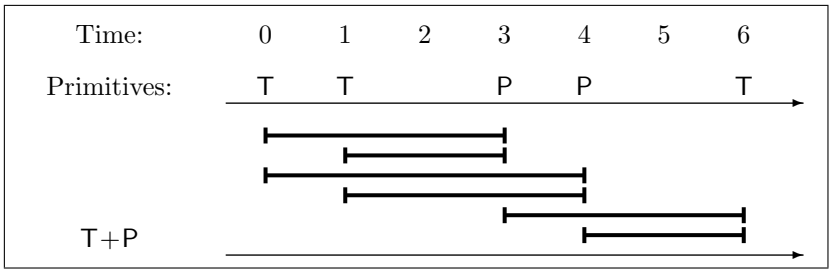

Figure 4.2: All occurrences of T+P.

Definition 4.9. *For two event streams, S and S', $res(S, S')$ holds if the following conditions hold:*

1. $S' \subseteq S$
2. $\forall s\ (s \in S\ \Rightarrow\ \exists s'(s' \in S' \wedge \text{start}(s) \leq \text{start}(s') \wedge \text{end}(s) = \text{end}(s')))$
3. $\forall s, s'\ ((s \in S' \wedge s' \in S' \wedge \text{end}(s) = \text{end}(s'))\ \Rightarrow\ s = s')$

Example 4.7. Figure 4.3 shows the detected instances of $(\mathsf{T}+\mathsf{P});\mathsf{B}$ in a particular scenario, and two valid restrictions S'_1 and S'_2 (i.e., both $res([\![(\mathsf{T}+\mathsf{P});\mathsf{B}]\!], S'_1)$ and $res([\![(\mathsf{T}+\mathsf{P});\mathsf{B}]\!], S'_2)$ hold). To see this, consider first the two instances with end time 4. The third criterion in the definition of *res* demands that only one of them is included in the restricted stream. The first and second criteria states that one of them must be included, and that we must in fact select the one that starts at time 2. In the same way, from the three instances with end time 6 we must include exactly one in the restricted stream, and it must be one of the two with start time 2. The choice between them, however, is arbitrary, and thus there are two valid restrictions, S'_1 and S'_2. ◇

For the user of the algebra, an important property of this policy is that at any time when there is one or more occurrences of A according to the semantics defined above, one of them will be detected (as ensured by the second criterion).

The fact that it is always an instance with maximum start time that is detected, is probably less significant to the user. However, this choice is crucial since it allows the restriction policy to be applied recursively to all subexpressions, without affecting the overall result. In order to

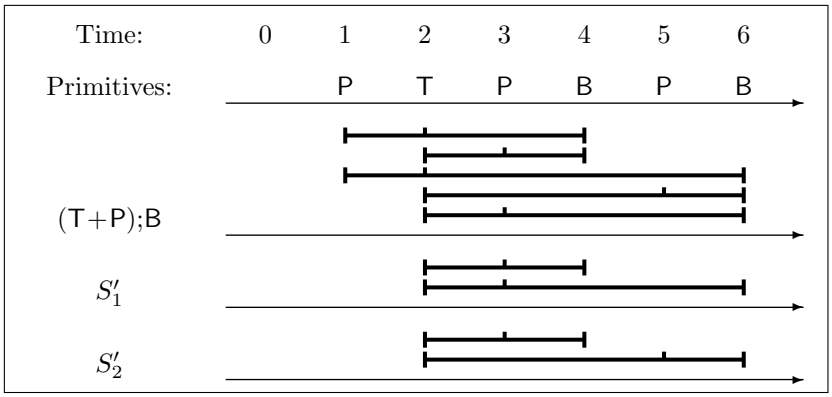

Figure 4.3: All occurrences of (T + P);B, and the two valid restricted streams S'_1 and S'_2.

get the desired efficiency, all parts of an expression must be detected in an efficient way, and this is possible if the restriction policy is applied to each subexpression. This would normally require a user of the algebra to understand how the restrictions in different subexpressions interfere with each other, and how they affect different operator combinations. To avoid this, the restriction policy has been designed to support the following theorem, which ensures that applying the restriction to all subexpressions gives a result which is valid also for the case when restriction is applied only at the top level. Thus, to the user, the restriction policy is applied only once to the whole expression, but a detection algorithm can freely apply it to the subexpressions as well.

Theorem 4.1. *If* $res(S, S')$ *and* $res(T, T')$ *hold, than for any event stream* U *and* $\tau \in \mathcal{T}$ *the following implications hold:*

- $res(\mathrm{dis}(S', T'),\ U) \quad \Rightarrow \quad res(\mathrm{dis}(S, T),\ U)$
- $res(\mathrm{con}(S', T'),\ U) \quad \Rightarrow \quad res(\mathrm{con}(S, T),\ U)$
- $res(\mathrm{neg}(S', T'),\ U) \quad \Rightarrow \quad res(\mathrm{neg}(S, T),\ U)$
- $res(\mathrm{seq}(S', T'),\ U) \quad \Rightarrow \quad res(\mathrm{seq}(S, T),\ U)$
- $res(\mathrm{tim}(S', \tau),\ U) \quad \Rightarrow \quad res(\mathrm{tim}(S, \tau),\ U)$

Proof. The proof can be found in Appendix A. □

Although a single stream may have several valid restrictions, they all share an important characteristic: They are equivalent with respect to instance start and end times.

Proposition 4.2. *If* $res(S, T)$ *and* $res(S, T')$ *then for each* $t \in T$ *there exists a* $t' \in T'$ *with* $\text{start}(t) = \text{start}(t')$ *and* $\text{end}(t) = \text{end}(t')$.

Proof. Since $T \subseteq S$, $t \in S$. By the second condition in the definition of *res*, there exists some $t' \in T'$ such that $\text{start}(t) \leq \text{start}(t')$ and $\text{end}(t) = \text{end}(t')$. We also have $t' \in S$, and thus there is some $t'' \in T$ such that $\text{start}(t') \leq \text{start}(t'')$ and $\text{end}(t') = \text{end}(t'')$. According to the third condition in the definition of *res* this implies $t = t''$, which means that we have $\text{start}(t) \leq \text{start}(t') \leq \text{start}(t)$ and thus $\text{start}(t') = \text{start}(t)$. □

4.3 Properties

To aid a user of this algebra, we present a selection of algebraic laws. These laws facilitate formal and informal reasoning about the algebra and a system in which it is embedded, and show to what extent the operators behave according to intuition. For this, we first define expression equivalence.

Definition 4.10. *For event expressions* A *and* B *we define* $A \equiv B$ *to hold if* $[\![A]\!]^{\mathcal{I}} = [\![B]\!]^{\mathcal{I}}$ *for any interpretation* $\mathcal{I}$.

Trivially, $\equiv$ is an equivalence relation. Moreover, the following proposition shows that it satisfies the substitutive condition, and thus defines structural congruence over event expressions.

Proposition 4.3. *If* $A \equiv A'$, $B \equiv B'$ *and* $\tau \in \mathcal{T}$, *then we have* $A \vee B \equiv A' \vee B'$, $A + B \equiv A' + B'$, $A;B \equiv A';B'$, $A - B \equiv A' - B'$ *and* $A_\tau \equiv A'_\tau$.

Proof. This follows directly from Definition 4.8. □

The laws presented later in this section identify expressions that are semantically equivalent with respect to the operator semantics, but in order to deal with resource limitations, we expect an implementation of the algebra to compute an event stream S such that $res([\![A]\!], S)$, rather than the full $[\![A]\!]$ stream. Since *res* is a predicate and not a function,

detecting A might potentially yield a different stream than detecting A', even when $A \equiv A'$. Consequently, it should be clarified to what extent the restriction policy affects expression equivalence.

Proposition 4.4. *If $A \equiv A'$ and res($[\![A]\!], S$), then res($[\![A']\!], S$).*

Proof. Since $A \equiv A'$ implies that $[\![A]\!] = [\![A']\!]$, this holds trivially. □

Thus, $A \equiv A'$ ensures that for any implementation consistent with the restriction policy, the detected occurrences of A is always a valid result for A' as well. Any reasoning based on the algebra semantics and the restriction policy, and not on the details of a particular detection algorithm, will be equally valid for equivalent expressions.

The next proposition ensures that although the detection of A and A' may not be exactly identical, they must be equivalent with respect to start and end times.

Proposition 4.5. *If $A \equiv A'$, res($[\![A]\!], S$) and res($[\![A']\!], S'$), then for any $s \in S$ there exists a $s' \in S'$ with* $\mathrm{start}(s) = \mathrm{start}(s')$ *and* $\mathrm{end}(s) = \mathrm{end}(s')$.

Proof. This follows straightforwardly from Proposition 4.2. □

The algebraic properties are given in the theorems below. Derived laws are indicated by an asterisk ($*$), and the proofs can be found in Appendix A. For instance frameworks with an construction operator that does not satisfy the commutativity and associativity requirements, all laws except number 3 (requires commutativity), 5 and 6 (require associativity) still hold. Note that the laws derived from these three laws (8, 13, 21, 30 and 32) hold anyway, since they can be proven individually.

Theorem 4.6. *For event expressions A, B and C, the following laws hold:*

1.	$A \vee A \equiv A$	6.	$A;(B;C) \equiv (A;B);C$
2.	$A \vee B \equiv B \vee A$	7.	$(A \vee B)+C \equiv (A+C) \vee (B+C)$
3.	$A+B \equiv B+A$	$*$8.	$A+(B \vee C) \equiv (A+B) \vee (A+C)$
4.	$A \vee (B \vee C) \equiv (A \vee B) \vee C$	9.	$(A \vee B);C \equiv (A;C) \vee (B;C)$
5.	$A+(B+C) \equiv (A+B)+C$	10.	$A;(B \vee C) \equiv (A;B) \vee (A;C)$

Theorem 4.7. *For event expressions A, B and C, the following laws hold:*

$$
\begin{aligned}
&11. \quad (A \vee B) - C \equiv (A - C) \vee (B - C)\\
&12. \quad (A + B) - C \equiv ((A - C) + B) - C\\
&*13. \quad (A + B) - C \equiv (A + (B - C)) - C\\
&14. \quad (A - B) - C \equiv A - (B \vee C)\\
&*15. \quad (A - B) - B \equiv A - B\\
&*16. \quad (A - B) - C \equiv (A - C) - B\\
&17. \quad (A;B) - C \equiv ((A - C);B) - C\\
&18. \quad (A;B) - C \equiv (A;(B - C)) - C
\end{aligned}
$$

Theorem 4.8. *For event expressions A and B, and $\tau \in \mathcal{T}$, the following laws hold:*

$$
\begin{aligned}
&19. \quad (A \vee B)_\tau \equiv A_\tau \vee B_\tau && 24. \quad (A;B)_\tau \equiv (A_\tau;B)_\tau\\
&20. \quad (A + B)_\tau \equiv (A_\tau + B)_\tau && 25. \quad (A;B)_\tau \equiv (A;B_\tau)_\tau\\
&*21. \quad (A + B)_\tau \equiv (A + B_\tau)_\tau && 26. \quad A \equiv A_\tau \quad \textit{if } A \in \mathcal{P}\\
&22. \quad (A - B)_\tau \equiv A_\tau - B && 27. \quad (A_\tau)_{\tau'} \equiv A_{\min(\tau,\tau')}\\
&23. \quad (A - B)_\tau \equiv (A - B_\tau)_\tau && *28. \quad (A_\tau)_{\tau'} \equiv (A_{\tau'})_\tau
\end{aligned}
$$

Finally, we introduce the notion of an empty event that never occurs, and laws related to this.

Definition 4.11. *Let the constant* $\mathbf{0}$ *denote the empty event, semantically defined as* $[\![\mathbf{0}]\!]^{\mathcal{I}} = \emptyset$ *for any interpretation* $\mathcal{I}$.

Theorem 4.9. *For an event expression A the following laws hold:*

$$
\begin{aligned}
&29. \quad \mathbf{0} \vee A \equiv A && 34. \quad \mathbf{0} - A \equiv \mathbf{0}\\
&*30. \quad A \vee \mathbf{0} \equiv A && 35. \quad A - \mathbf{0} \equiv A\\
&31. \quad \mathbf{0} + A \equiv \mathbf{0} && 36. \quad \mathbf{0};A \equiv \mathbf{0}\\
&*32. \quad A + \mathbf{0} \equiv \mathbf{0} && 37. \quad A;\mathbf{0} \equiv \mathbf{0}\\
&33. \quad A - A \equiv \mathbf{0} && 38. \quad \mathbf{0}_\tau \equiv \mathbf{0}
\end{aligned}
$$

Proof. These laws follow straightforwardly from the operator semantics and the definition of $\mathbf{0}$. □

Alternatively, $\mathbf{0}$ can be defined as shorthand for an expression $A - A$, where A is an arbitrary event expression (compare with law 33).

Chapter 5

Realisation and Resource Analysis

A primary ambition when designing the algebra has been to ensure that it can be implemented in such a way that bounds on memory footprint and processing time can be determined statically. As discussed in Section 4.2, the restriction policy is introduced to deal with resource problems associated with the declarative semantics. With this policy, most operators can be given bounded implementations straightforwardly. However, the following example illustrates why this is not the case for the sequence operator, and outlines how we address the problem.

Example 5.1. Consider an event expression $A;B$, where A and B are composite expressions. Figure 4.2 illustrates a scenario where at time 10, six previous occurrences of A has been detected. When b_1 is detected at time 12, the sequence operator semantics specifies that it can be matched with any of $a_1 \ldots a_4$ to form an instance of $A;B$. However, the restriction policy stipulates that the resulting instance should be one with maximum start time, and thus a_4 is the only valid choice.

The problem is that at time 10, we do not know which of the six A instances that will be the best match for some future B occurrence. In fact, we know that a_3 will not be used, since any B instance starting after the end of a_3 also starts after the end of a_2, and since the restriction policy requires that the combination with maximum start time is selected, a_2 will always be preferred over a_3. Each of the remaining five

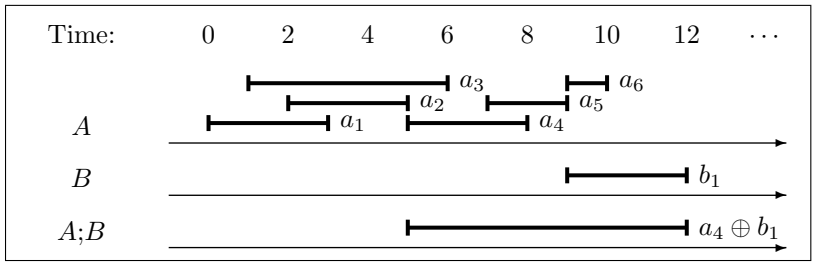

Figure 5.1: Graphical representation of Example 5.1.

A instances, however, may be the only valid alternative for some future B instance, and hence all of them have to be stored by a naive detection algorithm.

Our approach to this problem is to propagate not only full detections of B, but also the possible start times of future B occurrences, to the mechanism responsible for detecting the $A;B$ sequence. If, at time 10, we know that the start time of all future instances of B will be 6, 9 or greater than 10, then it suffices to store a_2, a_4 and a_6. Fortunately, the number of simultaneously active "possible start times" can be bounded, which allows a bounded memory implementation of the algebra. ◇

This chapter first presents an imperative detection algorithm based on this idea, and shows that this algorithm correctly implements the algebra semantics and the restriction policy. The algorithm is analysed with respect to time and memory complexity, and experiments are presented that investigate the actual worst case time and memory usage for randomly generated expressions.

5.1 Detection algorithm

Figure 5.2 presents a detection algorithm that, for a given event expression E, detects instances of an event stream S for which $res([\![E]\!], S)$ holds. The algorithm is executed once every time tick, i.e., once for each element in $\mathcal{T}$, and computes the current instance of E from the current instances of the primitive events, and from stored information about past occurrences. This time driven execution style simplifies the presentation and analysis of the algorithm, but it is clearly not an optimal strategy

in resource constrained systems where events occur rarely with respect to the granularity of the temporal domain. For this reason, Section 5.3 describes how this algorithm can be used in a more efficient, event driven setting.

Throughout this chapter, E denotes the event expression that is to be detected. The numbers $1 \ldots m$ are assigned to the subexpressions of E in an arbitrary bottom-up order, and we let E^i denote subexpression number i. Consequently, we have $E^m = E$ and $E^1 \in \mathcal{P}$. The symbol ε is used to represent a non-occurrence, and we define $\text{start}(\varepsilon) = \text{end}(\varepsilon) = -1$ to simplify the algorithm.

The variables used in the algorithm can be divided into three categories (see Table 5.1). *Persistent* variables store information that must be remembered from one time tick to the next in order to detect the event properly. Since each subexpression requires its own persistent variables, they are indexed from 1 to m. *Auxiliary* variables are used to pass information from a subexpression to its parent node in the expression tree. In particular, a_i is used to store the current instance of E^i, and thus a_m contains the output of the algorithm after each execution. The auxiliary variables are indexed in the same way as the persistent variables. Finally, there are *temporary* variables that are used locally within a single subexpression in a single tick. These are not indexed, indicating that the content is never used outside that scope.

Table 5.1: Variables used in the detection algorithm.

Category	**Variable**	**Type**	**Initial value**
Persistent	l_i, r_i	instance	ε
	Q_i	instance set	$\emptyset$
	t_i	time	-1
Auxiliary	a_i	instance	
	S_i	time set	$\emptyset$
Temporary	t	time	
	e, e'	instance	
	Q'	instance set	

```
for i from 1 to m
    if E^i ∈ P then
        if there is a current instance e of E^i then a_i := e
        else a_i := ε
    if E^i = E^j ∨ E^k then
        if start(a_j) ≤ start(a_k) then a_i := a_k else a_i := a_j
        S_i := S_j ∪ S_k
    if E^i = E^j + E^k then
        if start(l_i) < start(a_j) then l_i := a_j
        if start(r_i) < start(a_k) then r_i := a_k
        if l_i = ε ∨ r_i = ε ∨ (a_j = ε ∧ a_k = ε) then a_i := ε
        else if start(a_k) ≤ start(a_j) then a_i := a_j ⊕ r_i
            else a_i := l_i ⊕ a_k
        S_i := S_j ∪ S_k ∪ {start(l_i), start(r_i)}\{−1}
    if E^i = E^j − E^k then
        if t_i < start(a_k) then t_i := start(a_k)
        if t_i < start(a_j) then a_i := a_j else a_i := ε
        S_i := S_j
    if E^i = E^j ; E^k then
        e' := ε
        foreach e in Q_i ∪ {l_i}
            if end(e) < start(a_k) ∧ start(e') < start(e) then e' := e
        if e' ≠ ε then a_i := a_k ⊕ e' else a_i := ε
        Q' := ∅
        foreach t in S_k
            e' := ε
            foreach e in Q_i ∪ {l_i}
                if end(e) < t ∧ start(e') < start(e) then e' := e
            Q' := Q' ∪ {e'}
        Q_i := Q'
        if start(l_i) < start(a_j) then l_i := a_j
        S_i := S_j ∪ {start(e) | e ∈ Q_i ∪ {l_i}}\{−1}
    if E^i = (E^j)_τ then
        if end(a_j) − start(a_j) ≤ τ then a_i := a_j else a_i := ε
        S_i := S_j
```

Figure 5.2: The detection algorithm. For an event expression E, the content of a_m at the end of each time tick form an event stream $\mathcal{A}(m)$ which satisfies $res([\![E]\!], \mathcal{A}(m))$. Initially, $t_i = -1$, $l_i = r_i = \varepsilon$ and $S_i = Q_i = \emptyset$ for $1 \leq i \leq m$.

After executing the algorithm, the variable a_i contains the detected occurrence of E^i in the current tick, or ε if there is none. To connect this with the algorithm semantics, we define an event stream corresponding to each a_i variable.

Definition 5.1. *For* $1 \leq i \leq m$*, define*

$$\mathcal{A}(i) = \{e \mid e \text{ is the value of } a_i \text{ at the end of some time tick} \wedge\, e \neq \varepsilon\}$$

Thus, the output of the algorithm is the event stream $\mathcal{A}(m)$, and as established by Theorem 5.5 in the next section, this event stream satisfies $\mathit{res}([\![E]\!], \mathcal{A}(m))$.

The algorithm is designed for detection of arbitrary expressions, and the main loop selects dynamically which part of the algorithm to execute for each subexpression. For systems where the event patterns of interest are static and known at compile-time, the main loop can be unrolled and the top-level conditionals, as well as all indices, can be statically determined. Also, the assignments of S_i variables can be removed for all subexpressions except those occurring somewhere within the right-hand argument of a sequence operator. A concrete example of this is given in Figure 5.3.

if there is a current instance e of T **then** $a_1 := e$ **else** $a_1 := \varepsilon$
if there is a current instance e of P **then** $a_2 := e$ **else** $a_2 := \varepsilon$
if $\mathrm{start}(l_3) < \mathrm{start}(a_1)$ **then** $l_3 := a_1$
if $\mathrm{start}(r_3) < \mathrm{start}(a_2)$ **then** $r_3 := a_2$
if $l_3 = \varepsilon \vee r_3 = \varepsilon \vee (a_1 = \varepsilon \wedge a_2 = \varepsilon)$ **then** $a_3 := \varepsilon$
else if $\mathrm{start}(a_2) \leq \mathrm{start}(a_1)$ **then** $a_3 := a_1 \oplus r_3$
    else $a_3 := l_3 \oplus a_2$
if there is a current instance e of B **then** $a_4 := e$ **else** $a_4 := \varepsilon$
if $t_5 < \mathrm{start}(a_4)$ **then** $t_5 := \mathrm{start}(a_4)$
if $t_5 < \mathrm{start}(a_3)$ **then** $a_5 := a_3$ **else** $a_5 := \varepsilon$

Figure 5.3: Statically simplified algorithm for detecting $(\mathsf{T}+\mathsf{P})-\mathsf{B}$. Initially, $t_5 = -1$ and $l_3 = r_3 = \varepsilon$.

5.2 Algorithm correctness

In order to prove that this algorithm correctly implements the algebra semantics and the restriction policy, we first introduce a number of predicates that capture different correctness properties of the algorithm. We proceed by proving the correctness of a single operator at a single time tick, for each of these properties. The full correctness proof is organised as two nested inductions: an inner induction over the subexpressions of E, and an outer induction over time.

5.2.1 Correctness properties

To achieve bounded memory, the sequence operator requires some knowledge about what is stored in the persistent variables of its subexpressions. This information is propagated by the S_i variables, and the following predicate indirectly defines their meaning. Informally, it states that the start time of any detected non-instantaneous event was already propagated in the previous tick, and that the S_i variables are not updated with arbitrary values, only with the current time.

Definition 5.2. *Define pcorr(i, τ) to hold iff the following criteria hold:*

1. $a_i = \varepsilon \ \vee \ \text{start}(a_i) = \tau \ \vee \ \text{start}(a_i) \in S$
2. $\forall t \ (t \in S_i \Rightarrow (t = \tau \vee t \in S))$

where S was the content of S_i at the start of the current time tick.

The operators that require information about what has happened in the past, store this state information in the persistent variables r_i, l_i, t_i and Q_i. The following predicate defines what they should contain at the start of time tick τ.

Definition 5.3. *Define state(i, τ) as follows:*

- *For $E^i \in \mathcal{P}$, $E^i = E^j \vee E^k$ and $E^i = E^j_\tau$, state(i, τ) holds trivially.*
- *For $E^i = E^j + E^k$, state(i, τ) holds iff*
  - *l_i is an element in $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < \tau\} \cup \{\varepsilon\}$ with maximum start time; and*
  - *r_i is an element in $\{e \mid e \in \mathcal{A}(k) \wedge \text{end}(e) < \tau\} \cup \{\varepsilon\}$ with maximum start time.*

- *For* $E^i = E^j - E^k$, *state*(i, τ) *holds iff*
  - t_i *is the maximum element in* $\{\text{start}(e) \mid e \in \mathcal{A}(k) \wedge \text{end}(e) < \tau\} \cup \{-1\}$.
- *For* $E^i = E^j; E^k$, *state*(i, τ) *holds iff*
  - l_i *is an element in* $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < \tau\} \cup \{\varepsilon\}$ *with maximum start time; and*
  - *for each* $t \in S_k$ *such that* $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < t\}$ *is non-empty,* Q_i *contains an element with maximum start time from that set.*

The fact that the output of the algorithm at a single time tick is consistent with the restriction policy, is captured by what can be thought of as a pointwise restriction predicate, and a lemma that relates it to the ordinary restriction policy.

Definition 5.4. *For an event instance* e*, an event stream* S *and* $\tau \in \mathcal{T}$*, define* valid(e, S, τ) *to hold if:*

$$\Big(e \in S \wedge \text{end}(e) = \tau \wedge \neg\exists s(s \in S \wedge \text{end}(s) = \tau \wedge \text{start}(e) < \text{start}(s))\Big) \vee \Big(e = \varepsilon \wedge \neg\exists s(s \in S \wedge \text{end}(s) = \tau)\Big)$$

Lemma 5.1. *For an event stream* S *and event instances* $e_0, e_1, e_2, \ldots$ *such that* valid(e_τ, S, τ) *holds for any* $\tau \in \mathcal{T}$*, let* $S' = \{e_0, e_1, e_2, \ldots\} \backslash \{\varepsilon\}$*. Then* res$(S, S')$ *holds.*

Proof. By the definition of *valid*, it follows that $S' \subseteq S$. Next, take an arbitrary $s \in S$, and let $\tau = \text{end}(s)$. Since valid(e_τ, S, τ), we must have $e_\tau \neq \varepsilon$, and thus $e_\tau \in S'$. From the definition of *valid*, we know that $\text{start}(s) \leq \text{start}(e_\tau)$. We also have $\text{end}(e_\tau) = \text{end}(s)$, which means that the second requirement in the definition of *res* is satisfied. Finally, all elements in S' have different end times. Together, this implies that res(S, S') holds. □

The following correctness property represents that the detected instance, or non-occurrence, of E^i is correct with respect to the instances detected for the subexpressions.

Definition 5.5. *Define* $acorr(i, \tau)$ *as follows:*

- *For* $E^i \in \mathcal{P}$, $acorr(i, \tau)$ *holds iff* $valid(a_i, [\![E^i]\!], \tau)$
- *For* $E^i = E^j \vee E^k$, $acorr(i, \tau)$ *holds iff* $valid(a_i, \mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$
- *For* $E^i = E^j + E^k$, $acorr(i, \tau)$ *holds iff* $valid(a_i, \mathrm{con}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$
- *For* $E^i = E^j - E^k$, $acorr(i, \tau)$ *holds iff* $valid(a_i, \mathrm{neg}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$
- *For* $E^i = E^j ; E^k$, $acorr(i, \tau)$ *holds iff* $valid(a_i, \mathrm{seq}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$
- *For* $E^i = E^j_{\tau'}$, $acorr(i, \tau)$ *holds iff* $valid(a_i, \mathrm{tim}(\mathcal{A}(j), \tau'), \tau)$

5.2.2 Correctness results

Focusing first on the result of a single subexpression at a single time tick, we show that each of the three correctness properties hold under some given assumptions.

Lemma 5.2. *Assume that* $state(i, \tau)$ *held at the start of the current tick and that* $pcorr(n, \tau)$ *and* $acorr(n, \tau)$ *hold for all* $1 \leq n < i$. *Then* $state(i, \tau + 1)$, $pcorr(i, \tau)$ *and* $acorr(i, \tau)$ *hold after executing the loop body once.*

Proof. The proof can be found in Appendix A. □

The correctness lemma above is used in the inductive step of the two nested induction proofs over the expression and over time, respectively.

Lemma 5.3 (Inner induction)**.** *Let* τ *be the current time, and assume that for each* $1 \leq i \leq m$ $state(i, \tau)$ *held at the start of this tick. Then* $state(i, \tau+1)$ *and* $acorr(i, \tau)$ *holds for each* $1 \leq i \leq m$ *after executing the whole detection algorithm.*

Proof. In addition to to the assumption about *state*, assume that after executing the loop body $n - 1$ times, $pcorr(i, \tau)$ and $acorr(i, \tau)$ hold for all $1 \leq i < n$. As a base case, this clearly holds for $n = 1$. Then $state(n, \tau + 1)$, $pcorr(n, \tau)$ and $acorr(n, \tau)$ hold after loop iteration n, according to Lemma 5.2. By induction, the lemma holds. □

Lemma 5.4 (Outer induction)**.** *For any* i *such that* $1 \leq i \leq m$, *and any* $\tau \in \mathcal{T}$ $acorr(i, \tau)$ *holds after executing the algorithm at ticks* 0 *to* τ.

Proof. For the base case we see that $state(i, 0)$ holds in an initial state where $t_i = -1$, $l_i = r_i = \varepsilon$ and $Q_i = \emptyset$. For the inductive case: Assume that for some $\tau \in \mathcal{T}$, $state(i, \tau)$ holds at the start of tick τ. Then, according to Lemma 5.3, $state(i, \tau{+}1)$ and $acorr(i, \tau)$ holds after execution the algorithm, and thus $state(i, \tau{+}1)$ holds at the start of tick $\tau{+}1$. By induction over time the lemma thus holds for any $\tau \in \mathcal{T}$. □

So far, we have only shown that the result produced for E^i is correct with respect to the result produced by its subexpressions. Now, we take the final step and prove the correctness of the algorithm in the following theorem.

Theorem 5.5. *For any i such that $1 \leq i \leq m$, $res([\![E^i]\!], \mathcal{A}(i))$ holds.*

Proof. Assume that for some i, $res([\![E^n]\!], \mathcal{A}(n))$ holds for all $1 \leq n < i$. For the base case, this trivially holds for $i = 1$. According to Lemma 5.4, $acorr(i, \tau)$ holds at the end of tick τ. For $E^i \in \mathcal{P}$, we know from the definition of *acorr* that $valid(a_i, [\![E^i]\!], \tau)$ holds at the end of tick τ, and then Lemma 5.1 ensures $res([\![E^i]\!], \mathcal{A}(i))$. If $E^i = E^j \vee E^k$, the definition of *acorr* implies that $valid(a_i, \mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds at the end of tick τ, so by Lemma 5.1 we have $res(\mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$. According to Theorem 4.1, this together with the induction assumption that $res([\![E^j]\!], \mathcal{A}(j))$ and $res([\![E^k]\!], \mathcal{A}(k))$ hold (since $j < i$ and $k < i$), implies $res(\mathrm{dis}([\![E^j]\!], [\![E^k]\!]), \mathcal{A}(i))$ and thus $res([\![E^i]\!], \mathcal{A}(i))$. The proofs for the remaining operators are analogous. □

5.3 Algorithm improvements

To simplify presentation and correctness analysis, the algorithm uses set variables, and a time driven execution style was assumed where the algorithm is executed once every time instant. However, these design alternatives also have an impact on the efficiency of the algorithm, which must be addressed and resolved.

Considering first the issue of time triggered execution, we can see that in time ticks where no primitive events occur, none of the persistent variables are changed by the algorithm, and the a_i variables all become ε. In fact, this means that $\mathcal{A}(m)$ remains the same if the algorithm is executed only in ticks when at least one of the primitive events in E has occurred. Consequently, the algorithm presented here can be used with little or no changes also in an event driven setting where the execution

of the algorithm is triggered by primitive event occurrences rather than at each tick. If primitive events are non-simultaneous and always trigger the algorithm in the same order as they occur, the algorithm can be used without changes. Otherwise, some precautions must be taken to ensure that occurrences are processed in the right order.

This improvement could be taken further by processing only subexpressions that are affected by the current primitive event occurrences. The identification of what parts of the tree to consider could either be done statically with respect to the primitive events, or dynamically based also on the detection result of the subexpressions. The details of this optimisation technique is yet to be investigated, though, and it is further discussed as future work in Section 8.3.3.

Turning to the set variables, we notice that the worst part of the algorithm, from a complexity point of view, is the nested foreach constructs in the sequence part. This source of complexity can be avoided, without compromising the correctness of the algorithm, if the set variables S_i and Q_i are represented as ordered structures.

First, note that Q_i never contain fully overlapping instances. New values that are added to Q_i always come from l_i, and whenever l_i is updated, both the start and end time of the new instance is greater than those of the previous instance. Thus, if Q_i is ordered with respect to end times, it will also be ordered with respect to start time.

The time complexity of the S_i assignments is not affected by the ordered representation. For the sequence part of the algorithm, this follows from the fact that Q_i is ordered with respect to start times. The assignment of Q_i is done by means of a temporary set variable Q' that is populated by the best match, from the instances currently stored in Q_i and l_i, for each element in S_k. In the original detection algorithm, this is performed by the two nested foreach constructs shown in Figure 5.4, but when Q_i and S_k are ordered, it can be accomplished by a single pass over the two structures together, as shown in Figure 5.5. An array style notation is used for references to individual elements of an ordered structure, e.g., $S_k[1]$ for the first element of S_k.

5.4 Complexity analysis

Most parts of the algorithm are fairly straightforward to analyse with respect to time and memory usage, but we need to establish bounds on

```
foreach t in S_k
    e' := ε
    foreach e in Q_i ∪ {l_i}
        if end(e) < t ∧ start(e') < start(e) then e' := e
    Q' := Q' ∪ {e'}
```

Figure 5.4: Part of the original sequence operator algorithm.

```
Q_i := Q_i ∪ {l_i}
sp := length(S_k)
qp := length(Q_i)
while sp > 0 ∧ qp > 0
    t := S_k[sp]
    e := Q_i[qp]
    if end(e) < t then
        Q' := Q' ∪ {e}
        sp := sp − 1
    else qp := qp − 1
```

Figure 5.5: Improved version of the algorithm snippet in Figure 5.4 for the case when Q_i and S_k are ordered.

the set variables S_i, Q_i and Q'. For this, let $|X|$ denote the maximum size of a set variable X.

Proposition 5.6. *If* $E^i = E^j;E^k$ *then* $|Q_i| \leq |S_k|$*, otherwise* $|Q_i| = 0$*. We also have* $|Q'| = \max_{1 \leq i \leq m}(|Q_i|)$*.*

Proof. This follows straightforwardly from the assignments of Q_i and Q' in the algorithm. □

Proposition 5.7. *For any* i *such that* $1 \leq i \leq m$*, we have* $|S_i| <$ subexp(E^i) *where* subexp(A) *denote the number of subexpressions in* A*.*

Proof. In the base case $i = 1$, we have $E^i \in \mathcal{P}$ and thus $|S_i| = 0$ and subexp$(E^i) = 1$, which clearly satisfies the claim. For the inductive case we assume that $|S_n| <$ subexp(E^n) holds for all $1 \leq n < i$. If $E^i \in \mathcal{P}$ we

can repeat the proof for the base case. If $E^i = (E^j)_\tau$, then $|S_i| = |S_j|$, and since $j < i$, the assumption implies that $|S_j| < \text{subexp}(E^j)$. Thus, we have $|S_i| < \text{subexp}(E^j) < \text{subexp}(E^i)$. In the remaining cases where E^i is a binary operator applied to E^j and E^k, we have $j < i$ and $k < i$ and thus the assumption implies that $|S_j| \leq \text{subexp}(E^j) - 1$ and $|S_k| \leq \text{subexp}(E^k) - 1$. From the assignments of S_i in the algorithm, we see that $|S_i| \leq |S_j| + |S_k| + 2$ holds for all operators (for sequence, we use the fact that $|Q_i| \leq |S_k|$). Thus, $|S_i| \leq |S_j| + |S_k| + 2 \leq \text{subexp}(E^j) + \text{subexp}(E^k) < \text{subexp}(E^i)$, which concludes the proof. □

The memory and time complexity of the algorithm also depends on the particularities of the instance framework. Hence, we introduce the parameter ω to denote the maximum memory needed to store an instance in the current framework. An instance of a subexpression of E is constructed from at most $\lceil m/2 \rceil$ primitive instances (one from each leaf in the expression tree). Thus, assuming that primitive instances are of bounded size, and that the size of $a \oplus b$ is bounded whenever the size of a and b is, the instance size is bounded. For the time analysis, we assume that the time it takes to perform the $\oplus$ operation is proportional to ω, or asymptotically lower. As previously, m denotes the number of subexpressions in E.

Theorem 5.8. *The memory complexity of the algorithm is* $\mathrm{O}(m^2\omega)$.

Proof. Since $\text{subexp}(E^i) \leq m$ for any $1 \leq i \leq m$, it follows from Propositions 5.6 and 5.7 that $|Q'| \leq m$ and that $|S_i| \leq m$ and $|Q_i| \leq m$ for any $1 \leq i \leq m$. This means that the algorithm stores at most $\mathrm{O}(m^2)$ instances and time values. □

Theorem 5.9. *The time complexity of the algorithm is* $\mathrm{O}(m^2\omega)$.

Proof. The algorithm performs m iterations of the main loop, each iteration executing one of the operator specific parts of the loop body. Only the code for the sequece operator contains loop structures, so for the other operators the primary source of complexity are the assignments of the set variables S_i, and they can be performed in $\mathrm{O}(|S_i|\omega)$, also when the S_i variables are ordered. For the sequence operator, there are two loop structures, each with a body that runs in $\mathrm{O}(\omega)$ time. The foreach loop iterates $|Q_i| + 1$ times, and the while loop $|Q_i| + 1 + |S_k|$ times. Finally, the set assignment can be performed in $\mathrm{O}(|S_i|\omega)$ time when Q_i is ordered with respect to start time. Altogether, since Propositions 5.6

and 5.7 ensures that $|Q_i|$, $|S_i|$ and $|S_k|$ are less than or equal to m, the code for each operator can be executed in $O(m\omega)$ time. Thus, the time complexity of the whole algorithm is $O(m^2\omega)$. □

Example 5.2. In the simple framework of Example 4.2, ω is a constant factor, and thus the time and memory complexity are $O(m^2)$. In the framework of Example 4.3, the instance size is bounded by $\lceil m/2 \rceil$, and thus the memory and time complexity of the algorithm are $O(m^3)$. ◇

5.5 Memory and execution time analysis

The complexity analysis presented in the previous section illustrates how the resource demands of the detection algorithm, in terms of memory and time, increase as the size of expressions increases. This indicates the general usefulness of the proposed algebra for large expressions, but it does not provide much insight into the resource demand associated with the detection of a particular pattern. For example, the complexity analysis regards the worst possible expression of a given size, but most expressions have significantly lower resource demands since different operator combinations contribute very differently to the overall time and memory usage. The amount of memory required to detect a certain pattern is of interest for embedded systems where resources are limited, but also for safety-critical systems where the absence of errors caused by memory shortage must be guaranteed statically. Systems with real-time constraints, e.g., where an external event has to be responded to within a given time, require information about the execution time of different parts of the system in order to guarantee that these constraints are satisfied. In particular, schedulability and timeliness analysis typically assume that the *worst case excecution time* (WCET) is known for each critical activity in the system [28]. For details on WCET analysis, the reader is referred to the recent survey by Wilhelm et al. [153].

There are two ways in which an event pattern can be analysed with respect to the time and memory needed to detect it: via synthesised code or based directly on the event expression. In systems where event patterns are static, and specified during the development of the system, specific detection code can be synthesised for each expression, as discussed in Section 5.1. This code can be analysed with standard analysis tools to acquire information about memory footprint and execution

time. The generated detection code does not utilise any dynamic memory management, neither by explicit memory allocation, nor by function calls or parameter passing via the runtime stack. Furthermore, the code is characterised by a very simple control flow. For example, there are no subroutine or function calls, and all loops are trivially bounded by the size of some static data structure. This means that the code does not contain any of the constructs that Puschner and Koza [125] identify as the main obstacles when determining the execution time of a program. Thus, existing techniques and standard tools for execution time analysis, e.g., SWEET [154], aiT [2] or Bound-T [24] should be applicable.

Alternatively, abstract notions of memory footprint and worst case execution time can be derived directly from the expression and the instance framework. Memory can be quite closely represented by the number of basic type variables, i.e., integers, booleans, etc., that are used by the detection algorithm, abstracting from the exact amount of memory needed to store them on a particular hardware. Similarly, but less accurately, worst case response time can be expressed as the number of assignments, comparisons, arithmetical operations, etc., that are executed in the worst case. This alternative is clearly the only option for systems where patterns are created dynamically, and where runtime decisions are made based on the resource demands of the patterns, e.g., which node in a distributed system that should handle the detection of a recently created pattern. It can also be useful for systems with static patterns, either because the target hardware has not yet been determined (or it is not covered by the available analysis tools), or to provide quick estimates that can be revised once the code has been generated.

Figure 5.6 presents parts of an analysis algorithm that computes abstract estimates of the memory footprint and worst case execution time of a given event expression, and the whole algorithm is given in Appendix B. If analyse(E) returns $\langle m, t \rangle$, this means that the detection of E requires m memory units and takes t units of time.

The algorithm assumes that time instants, array indices and integers all require one memory unit, and that sets are represented straightforwardly by ordered structures with direct access to individual elements. For the time analysis, it is assumed that comparisons, arithmetic operations and assigning a time instant variable take one time unit. The time it takes to assign an instance variable is the same as the size of that variable, and a set assignment $S' := S$ takes $1 + |S| * s$ time units, where s is the time it takes to assign a single element of S. In addi-

$$
\begin{array}{l}
\text{analyse}(E) = \langle m+1,\ t+2\rangle \\
\quad \text{where} \\
\quad \langle s, i, m, t\rangle = \text{analyse\_aux}(E,\ \text{false}) \\
\\
\text{analyse\_aux}(E^j \vee E^k,\ sn) = \langle s, i, m, t\rangle \\
\quad \text{where} \\
\quad \langle s_j, i_j, m_j, t_j\rangle = \text{analyse\_aux}(E^j,\ sn) \\
\quad \langle s_k, i_k, m_k, t_k\rangle = \text{analyse\_aux}(E^k,\ sn) \\
\quad \text{if } sn \text{ then } s = s_j + s_k \text{ else } s = 0 \\
\quad i = \text{unionsize}(i_j,\ i_k) \\
\quad m = m_j + m_k + 1 + s + i \\
\quad t = t_j + t_k + 5 + s + i \\
\\
\text{analyse\_aux}(E^j + E^k,\ sn) = \langle s, i, m, t\rangle \\
\quad \vdots
\end{array}
$$

Figure 5.6: Time and memory analysis algorithm. If analyse(E) returns $\langle m, t\rangle$, this means that the detection of E requires m memory units and takes t units of time.

tion to the complexity improvement described in the previous section (see Figure 5.5), we assume that the S_i variables are assigned only in subexpressions occuring within the right-hand argument of a sequence operator.

For example, consider a disjunction operator, and let i and s denote the instance size and size of S_i for this subexpression. Then the operator requires i units of memory for the a_i variable, and $1+s$ for the S_i variable (s for the stored time instants, and one extra to store the current size of S_i). Thus, the algorithm adds $1 + s + i$ to the memory usage of the subexpressions, for a disjunction operator. When the s variable is computed, the algorithm takes into account the optimisation of S_i variable assignments, as represented by the boolean parameter *sn*. On the top level, s is false, and it becomes true only when entering the right-hand subexpression of a sequence.

To account for different instance frameworks, and the type system of the underlying language, the following analysis primitives are used in the algorithm:

Definition 5.6.

- primsize(A) *denotes the maximum instances size of the primitive event* A.
- compsize(x, y) *denotes the size of* $a \oplus b$ *when* x *and* y *are the sizes of* a *and* b*, respectively.*
- unionsize(x, y) *denotes the size of an instance of* $A \vee B$ *when the size of* A *and* B *instances are* x *and* y*, respectively.*

5.5.1 Experiments

Based on the analysis algorithm, we have conducted a few experiments to investigate the resource requirements of the algorithm in more detail. Expressions containing m subexpressions were created randomly, with equal probability for the five operators to occur, and with a random structure. For each expression, the memory footprint and the worst case execution time were analysed, assuming first the simple instance framework from Example 4.2, where no additional data is associated with occurrences, and then for the framework with values, defined in Example 4.3. Each m value is represented by 10.000 random expressions, and the 95% confidence intervals for the mean values are less than 2% of the y-value for all points.

Experiment 1: For the memory analysis, we make the following assumptions: In the simple framework, all instances are represented by just the start and end time, and thus require 2 memory units, which is represented by the following analysis primitives:

$$\text{primsize}(A) = \text{compsize}(x, y) = \text{unionsize}(x, y) = 2$$

In the value framework, we assume that all primitives have value domains with elements that can be represented in a single memory unit. Thus, primitive instances require 3 units (time, id and value), and the size of a composite instance $a \oplus b$ is the summed size of a and b. For disjunctions, we assume that the instance size equals the largest instance size of the two constituent events plus an additional unit of memory, as would be the case if a discriminated union type [67] is used in a C implementation. Altogether, this is captured by:

$$\text{primsize}(A) = 3,\ \text{compsize}(x, y) = x + y,\ \text{unionsize}(x, y) = \max(x, y) + 1$$

Figures 5.7 and 5.8 show the mean and maximum memory footprint of the sample expressions, for different m values. The experiment shows that the detection algorithm memory usage is fairly low, also for complex expressions. The detection of an average expression consisting of 51 subexpressions requires less than 250 units of memory in the simple framework and roughly the double when all primitives carry values. Over the whole experiment, the maximum value is approximately twice as high as the average for the simple framework, and four times as high in the value framework. Although the actual worst case might be significantly higher than the maximum within the samples of 10.000 expressions investigated in the experiment, this indicates that expressions with high memory demand are very rare. ◇

Experiment 2: The execution time analysis is based on the same analysis primitives as the memory experiment above. Figures 5.9 and 5.10 show the mean worst case execution time, as well as the maximum, for each m value. The detection of an average expression consisting of 51 subexpressions takes less than 650 time units in the simple framework, and less than 980 in the value framework. As for memory, we note that the difference between average and maximum is relatively small. For the simple framework, maximum is approximately three times higher than average, and in the value framework it is four times higher. ◇

These experiments show that, although there might exist expressions with very high resource demands, these are very rare. None of the investigated expressions have memory footprint and execution time values that prevent them from being used in an embedded setting. Finally, it should be pointed out that the second experiment concerns worst case execution time. The execution time of an average tick depends on the actual arrival frequencies and patterns of the primitive events.

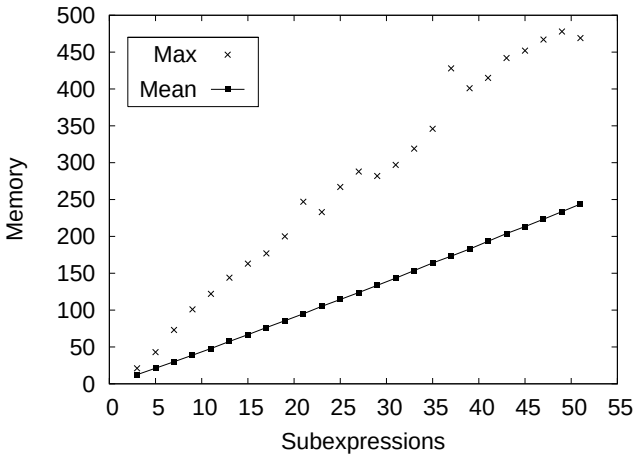

Figure 5.7: Memory usage in the simple framework.

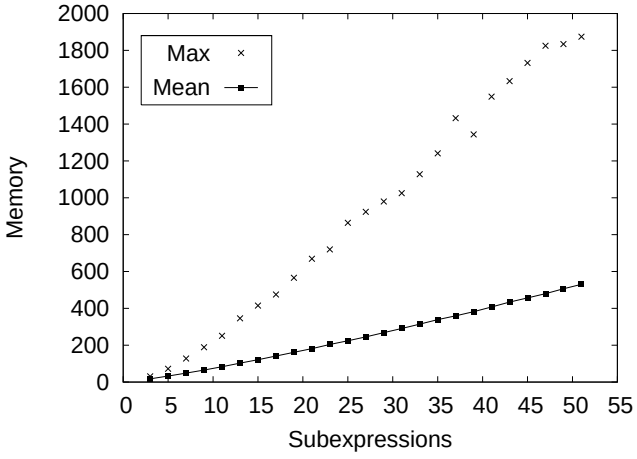

Figure 5.8: Memory usage in the value framework.

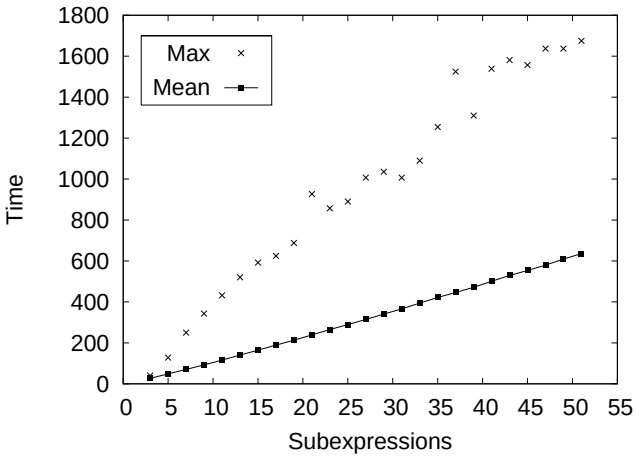

Figure 5.9: Worst case execution times in the simple framework.

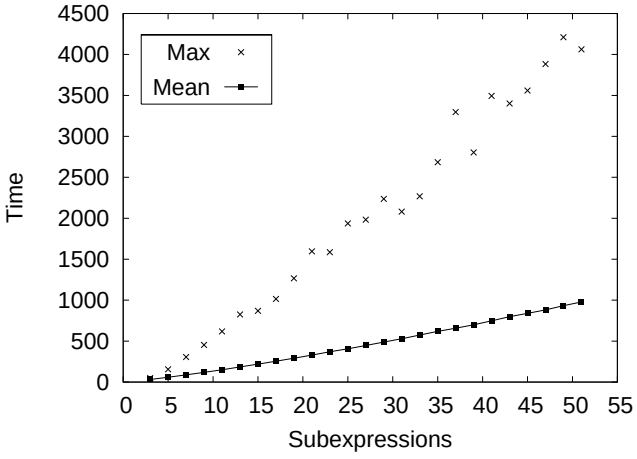

Figure 5.10: Worst case execution times in the value framework.

Chapter 6

Event Pattern Triggered Tasks

An embedded system can respond to external events either by periodically checking at predetermined points in time whether the event has occurred, or by setting up a task that is explicitly activated by the event in question [96, 156]. Generally speaking, the time triggered alternative results in more deterministic system behaviour, which is typically desirable for analysis purposes. However, for situations that require a quick response but occur relatively rarely, the time triggered approach can be prohibitively inefficient [28].

In this chapter, we address situations where the system should react to some complex pattern of event occurrences, rather than to individual occurrences. We introduce the concept of *pattern triggered tasks*, and show how they can be incorporated into an ordinary scheduling framework of periodic and sporadic tasks, which includes providing timeliness analyses for such mixed task sets.

6.1 Triggering tasks by patterns

A straightforward way to construct an embedded system that reacts to a particular event pattern would be to include a designated task for this reaction, and make this task responsible for carrying out the response under the right conditions. In an event triggered setting, the task would

be executed periodically, at a frequency determined by the urgency of the response and the available resources. If the system is event triggered, the task can be activated by the individual primitive events that are part of the pattern.

A drawback of this straightforward approach when used in an event triggered system is that the execution time of the task varies a lot, since the response code is only executed when the full pattern is detected. If pattern occurrences are rare compared to the occurrences of individual primitive events, analysis techniques based on a single WCET value will be very pessimistic, meaning that the system must be significantly underutilised in order to statically guarantee timeliness of all tasks. Splitting the task into two tasks, one that is responsible for pattern detection and one that carries out the response, allows more accurate resource usage estimates, but does not solve the problem. With no information about the pattern, we must assume that in the worst case any event occurrence results in a detection of the full pattern, and hence, that the response task executes as often as the detection task.

We propose a task model where triggering patterns are defined explicitly rather than implicitly in the task code. This means that the patterns are available for analysis, in particular to establish how frequently the pattern can occur in the worst case, which allows a more efficient use of resources.

In the following presentation, triggering patterns are specified by event expressions from our algebra. The algebra allows the developer to construct a declarative specification of the situations that trigger a response, which might provide better support for reasoning about the overall system behaviour than a procedural approach. As described in Chapter 5, efficient detection code with bounded worst case memory footprint and execution time can be automatically synthesised from any event expression in the algebra. However, the concept of pattern triggered tasks, and the proposed schedulability analysis for such tasks, can be used with other pattern specification techniques as well, as long as they provide the same type of pattern occurrence rate information. This is further discussed in Section 8.3.4.

6.1.1 Task model and assumptions

We assume an event triggered system, and the task model includes both periodic tasks and tasks that are triggered by a particular pattern of

sporadic events. For simplicity, ordinary sporadic tasks are treated as a special case of pattern triggered tasks where the pattern consists of just a single event. In our task model, a task is characterised by the following parameters:

- **Worst case execution time (C_i).** The longest time it could take to execute the code of the task, assuming that it is not interrupted.
- **Relative deadline (D_i).** The time, relative to activation, when the task must be finished.
- **Period (T_i).** The time between two consequitive activations of a periodic task.
- **Event expression (E_i).** A specification of the situation under which a pattern-triggered task should be activated.

We also make the following assumptions about the system:

- **Preemptive scheduling.** A task can be interrupted by the scheduler at any point during its execution, for example to allow another task to execute. Later, the scheduler can resume execution of the interrupted task.
- **Hard deadlines.** A task that does not meet its deadline is assumed to be of no use, or possibly harmful, to the system.
- **No self-suspension.** Tasks execute until completion, unless preempted by another task.
- **No kernel overhead.** The overhead due to scheduling activities, task switches, etc. is assumed to be negligible.
- **Independent tasks.** Tasks are not subject to precedence constraints, and do not share resources.

Example 6.1. As a running example, we consider a system with three tasks, two of which are periodic. The third task is triggered by the event expression $(A;B)+C$. The basic parameters of this example task set is presented in Table 6.1. ◇

Table 6.1: The task set from Example 6.1.

Task ID	C_i	T_i	D_i	E_i
T1	10	50	30	–
P2	20	–	100	$(A;B)+C$
T3	30	200	200	–

6.1.2 Realisation

Event pattern detection could be provided as a service in the underlying operating system, in which case realisation of pattern triggered tasks would be straightforward. For use in standard real-time operating systems, where pattern detection is not provided, there are two major alternatives, depending on how complex the event pattern is. For relatively simple patterns, meaning that the execution time of the corresponding detection algorithm is sufficiently short, pattern detection can be performed by the interrupt handler that is invoked when an interrupt signal arrives. Interrupt handlers typically execute at a higher priority than ordinary tasks, independently of scheduling policy, etc. Thus, the worst case interference from interrupt handling must be taken into account in the schedulability analysis, and having interrupt handlers with long worst case execution time may cause critical tasks to become non-schedulable. The impact from interrupt handling on schedulability has been addressed by Gonzalez Harbour et al. [71] and by Jeffay and Stone [79], for fixed and dynamic priority scheduling, respectively.

Alternatively, detection can be performed at task level, which means that it is subject to the same scheduling policy as other tasks. Rather than performing pattern detection, the interrupt handler only activates all tasks that are triggered by an event expression that includes the event in question, providing the ID of the primitive event, a time stamp and any additional data associated with the occurrence, as a part of the activation. When selected for execution by the scheduler, a pattern triggered task first executes the event detection mechanism, based on the event ID and time stamp received from the interrupt handler at activation. The code for this phase is generated automatically from the event expression, based on the detection algorithm from Section 5.1.

If the detection algorithm signals a successful detection of the whole pattern, the task proceeds with executing the response code, otherwise it terminates.

The reason for including the conceptually different activities of event detection and response in the same task is related to the fact that the original deadline of a pattern triggered task is relative to the pattern occurrence, not the time at which the pattern is detected and the response task is released [21]. Consequently, the execution of a response should not be delayed by the processing of primitive instances that occur later than the instance that caused the whole pattern occurrence, and thus triggered the response. Although this could be achieved also if detection and response were handled as separate tasks, by means of task properties such as deadlines and priorities, the analysis would have to take the temporal relation between them into consideration, leading to more complicated and possibly less exact analysis.

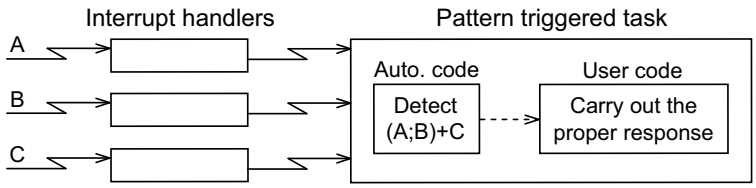

Figure 6.1: Task level realisation of a task triggered by the pattern $(A;B)+C$.

Figure 6.1 shows the realisation of the pattern triggered task from the previous example. The interrupt handler associated with each primitive event simply activates the task, providing the ID of the event and a time stamp. When selected for execution by the scheduler, the task executes the detection algorithm for $(A;B)+C$ and, if a pattern occurrence was detected, executes the proper reaction to the pattern.

Henceforth, we assume that detection is performed at task level. The main motivation is that there are no deadlines associated with the detection as such, only with the combined activity of detection and reacting to a detected occurrence of the pattern [21]. Performing pattern detection at interrupt priority potentially interferes with critical tasks without improving the response time of the desired reaction relative to the occurrence of the pattern.

6.1.3 Scheduling and schedulability

The role of a scheduler is to decide when to execute the different tasks, so as to satisfy any constraints on precedence, timeliness, etc. The aim of schedulability analysis is to determine whether a task set is schedulable under a particular scheduling policy, meaning that there are no circumstances under which a task violates its constraints.

In general, providing such guarantees require knowledge about the resources, in particular processor time, each task may demand over time. For periodic tasks, this is provided by period and WCET, but for aperiodic activities to be included in the guarantee, some assumptions have to be made about the environment to limit the frequency of the concerned events. For example, by assuming a minimum interarrival time for the events that activates a task, in which case the task is classified as *sporadic*.

Similarly, for event pattern triggered task to be analysable, we have to be able to make assumptions about the frequency of the primitive events that are part of the pattern. If no such information is available, it is difficult to guarantee the timeliness of these tasks or any task that may be affected by them. In such a scenario, pattern triggered tasks can only be served on best-effort basis with no guarantees, although there are ways to ensure that they do not interfere with the timely execution of periodic tasks, e.g., using some server based scheduling method. A survey of such techniques is given by Buttazzo [30].

We focus on the case when timely execution of pattern triggered tasks must be guaranteed as well as that of periodic tasks, and thus assume that the primitive events are *sporadic*, i.e., that the minimum interarrival time for each primitive event is known.

Definition 6.1. *The minimum interarrival time of primitive events is represented by the function* $\mathrm{mint} : \mathcal{P} \to \mathbb{Z}^+$. *An interpretation* $\mathcal{I}$ *is consistent with a* mint *function if for all* $A \in \mathcal{P}$, *two elements* a *and* a' *in* $\mathcal{I}(A)$ *satisfy* $\mathrm{mint}(A) \leq |\mathrm{end}(a') - \mathrm{end}(a)|$ *whenever* $a \neq a'$.

The occurrences of a pattern are in general not sporadic even though the primitive events are. For a simple example, consider two sporadic events A and B. Regardless of the minimum interarrival times of the two events, an occurrence of A can be immediately followed by an occurrence of B, resulting in two occurrences of the pattern $A \vee B$ separated by just a single clock tick.

Nevertheless, the frequency at which a pattern occurs can be bounded, but in a more general way than with a single minimum interarrival time value. For example, we can safely state that in any interval of length $\min(\text{mint}(A), \text{mint}(B))$, there can be at most two occurrences of the pattern $A \vee B$. This resembles the concept of *bursty aperiodic tasks* [83] which are triggered by events that can occur arbitrarily close in time, but which are bounded by a constraint specifying that there can be at most n occurrences within any interval of length l. For patterns, however, it is possible to be more specific than the two parameters of bursty tasks permit. First, we note that a pattern always occur at the same time as one of the constituent primitive events. In fact, only a subset of the constituent events can directly result in an occurrence of the pattern. For example, $A;B$ always occur at the same time as some B occurrence. The following definition formalises this idea.

Definition 6.2. *Let* $\text{prim}(E)$ *denote the primitive events of* E*, and* $\text{term}(E)$ *the terminating primitive events of* E*, defined as follows:*

$$\begin{aligned}
\text{term}(A) &= \{A\} \quad \text{if } A \in \mathcal{P} \\
\text{term}(A \vee B) &= \text{term}(A) \cup \text{term}(B) \\
\text{term}(A + B) &= \text{term}(A) \cup \text{term}(B) \\
\text{term}(A;B) &= \text{term}(B) \\
\text{term}(A - B) &= \text{term}(A) \\
\text{term}(A_\tau) &= \text{term}(A)
\end{aligned}$$

Proposition 6.1. *For any event expression* E*, interpretation* $\mathcal{I}$ *and event instance* e *such that* $e \in [\![E]\!]^{\mathcal{I}}$*, there exists a primitive event* $A \in \text{term}(E)$ *and an event instance* $a \in \mathcal{I}(A)$ *with* $\text{end}(e) = \text{end}(a)$*.*

Proof. This follows from the algebra semantics, specified in Definitions 4.7 and 4.8. □

Definition 6.3. *Let* $\text{wcet}(E)$ *denote the worst case execution time associated with the detection of* E*.*

The worst case detection execution time $\text{wcet}(E)$ can be derived either from the generated code using standard WCET analysis tools (e.g., SWEET [154], aiT [2] or Bound-T [24]) or by instantiating an abstract WCET estimate derived directly from the expression, expressed as the number of assignments, comparisons, arithmetical operations, etc., that are executed in the worst case. This is further discussed in Section 5.5.

The central idea in the schedulability analysis of systems with pattern triggered tasks is that a task responding to occurrences of a pattern E requires the same amount of computation resources over time as a particular set of sporadic tasks. Conceptually, we view those task instances that respond to pattern occurrences that was terminated by an occurrence of a particular primitive event A as instances of a separate task that those terminated by B.

Definition 6.4. *Let Γ be the original task set, and define the auxiliary taskset Γ^{aux} as the smallest set of tasks such that*

- *all periodic tasks in Γ are in Γ^{aux};*
- *for each pattern triggered task $t_i \in \Gamma$, and each $A \in \text{prim}(E_i)$, Γ^{aux} contains a sporadic task t_k with $T_k = \text{mint}(A)$, $D_k = D_i$ and*

$$C_k = \begin{cases} \text{wcet}(E_i) + C_i & \text{if } A \in \text{term}(E_i) \\ \text{wcet}(E_i) & \text{if } A \notin \text{term}(E_i) \end{cases}$$

Example 6.2. In order to analyse the task set from Example 6.1, we make the following assumptions about minimum interarrival times and worst case detection execution time: $\text{mint}(A) = 60$, $\text{mint}(B) = 70$, $\text{mint}(C) = 200$ and $\text{wcet}((A;B)+C) = 5$.

Note that with the straightforward approach described initially, where pattern detection is performed implicitly within the task code, this task set is not schedulable even if the detection overhead is disregarded. During any time interval of length 4200, 84 instances of T1 and 21 instances of T3 are released for execution. Furthermore, there can be 70, 60 and 21 occurrences of A, B and C, respectively, potentially triggering 151 instances of P2. Thus, the total amount of execution that is released during the interval is $84 * 10 + 21 * 30 + 151 * 20 = 4490$. If 4490 units of computation can be released in any interval of length 4200, the task set is clearly not schedulable.

Table 6.2 depicts the auxiliary task set, constructed according to Definition 6.4. Since $\text{prim}((A;B)+C) = \{A,\ B,\ C\}$, the pattern triggered task will result in three sporadic tasks in the auxiliary taskset, with minimum interarrival times given by these three primitive events, respectively. From $\text{term}((A;B)+C) = \{B,\ C\}$ it follows that the WCET of the original response task should be included only in those two tasks, i.e., in t_3 and t_4. ◇

Table 6.2: The auxiliary task set from Example 6.2.

From task	t_i	C_i	T_i	D_i
T1	t_1	10	50	30
P2	t_2	5	60	100
	t_3	25	70	100
	t_4	25	200	100
T3	t_5	30	200	200

The tasks in the auxiliary task set are either periodic or sporadic, which means that existing schedulability analysis theory can be applied. The remaining sections investigate this in more detail for two important scheduling approaches, based on fixed and dynamic priorities, respectively. The basic idea, though, should be possible to utilise in other scheduling frameworks as well.

6.2 Fixed priority scheduling

In fixed priority scheduling (FPS), each task is assigned a priority at development time. At runtime, the executing task is always one with highest priority from those that are currently available for execution.

For periodic tasks, priorities are often assigned according to the *rate monotonic* principle, meaning that a task with shorter period should have a higher priority. When the task set includes tasks that have a deadline shorter than the period, a *deadline monotonic* priority assignment can be used instead, where priorities are assigned based on deadlines rather than periods. Priorities can also be assigned in a way as to avoid certain runtime scenarios or to achieve some other goal, for example to mimic the execution of a given offline schedule, or to minimise the number of preemptions [46].

Liu and Layland [93] showed that under certain assumptions, e.g., that tasks are independent and that $D_i = T_i$, rate monotonic is an optimal fixed priority scheduling policy, meaning that any task set that is schedulable by some fixed priority assignment can be scheduled by rate monotonic as well. A similar proof for deadline monotonic is provided by Leung and Whitehead [89].

Joseph and Pandya [81] introduced *response time analysis*, a technique for establishing the worst case response time for tasks under FPS. They considered a relatively simple task set, where tasks are independent, have deadlines less than or equal to the period, etc, and their work has later been extended in various ways in order to avoid these restrictions, for example by Burns et al. [27] to account for scheduling overhead, and by Sha et al. [138] to cover tasks that communicate via shared resources. In particular, Tindell et al. [147] describe how the standard response time analysis technique can be extended to deal with the case when deadlines may be greater than the period. Since this is the case in the analysis of pattern triggered task, a brief overview of their approach is given.

For the simple task set in the standard response time analysis version, the worst case response times occur when all tasks are released simultaneously. For the case when several instance of a task can be active at the same time, however, the impact from such a simultaneous release can sometimes be higher on a subsequent instance than on the instances that were released together.

To account for this, let $L(P_i)$ denote the level P_i busy period, i.e., the maximum interval during which the processor continuously executes tasks with priority higher or equal to P_i. The precise definition of busy period, and an efficient algorithm for computing it, can be found for example in the work by George et al. [63].

Assuming that one instance of each task is released at the start of the busy period, and that subsequent instances are released with minimum interarrival time, the instances of a task t_i that arrive within the level P_i busy period will arrive at times qT_i, relative to the start of the busy period, for $q = 0, 1, \ldots, \lfloor L(P_i)/T_i \rfloor$. These are the instances that must be investigated in order to surely capture the worst case response time. For each of them, the response time is computed, taking into account the impact from higher priority task and from previous instances of t_i. The worst of these response times is the worst case response time of the task.

6.2.1 Pattern triggered tasks under FPS

For fixed priority scheduling, we extend the task model from Section 6.1.1 with an additional parameter:

- **Priority (P_i).** The static priority level of the task.

We also make the following assumption about the system:

- **FIFO.** Task instances that have the same priority are served in the same order as they are released. Ties are broken arbitrarily.

In a situation when several primitive events occur before the first has been processed, several instances of a pattern triggered task will be active at the same time. Since they have the same priority, the FIFO assumption is required to ensure that the primitive events are processed in the correct order.

For the schedulability analysis, we let the definition of Γ^{aux} cover priority as well, so that a sporadic tasks t_k in Γ^{aux} that was generated by a pattern triggered tasks t_i in Γ inherits the priority, i.e., $P_k = P_i$.

Lemma 6.2. *If Γ^{aux} is schedulable under FPS, then Γ is schedulable under FPS.*

Proof. We show that any sequence of Γ task arrivals can be mirrored by a sequence of Γ^{aux}task arrivals with the same arrival times, priorities and execution times. Since the periodic tasks are the same in both tasksets, the arrivals of periodic tasks in Γ can be mirrored by identical arrivals in Γ^{aux}. For the pattern triggered tasks, we note that every instances of the pattern triggered task t_i from Γ is triggered by an occurrence of some event in $\text{prim}(E_i)$. Thus, the activation times of the pattern triggered task can be mirrored by activations of the corresponding sporadic tasks in Γ^{aux}.

Next, consider an individual instance of t_i, triggered by an occurrence of the event A. If $A \notin \text{term}(E_i)$, then no occurrence of A can result in a full occurrence of the pattern, according to Proposition 6.1, and thus the t_i instance only executes the detection algorithm and not the response, which is consistent with the WCET of t_k, which is $C_k = \text{wcet}(E_i)$ when $A \notin \text{term}(E_i)$. For events in $\text{term}(E_i)$, Γ^{aux} safely approximates Γ by assuming that they always result in a full occurrence of the pattern. Clearly, the execution time of the t_i instance can not exceed $\text{wcet}(E_i) + C_i$.

Altogether, this means that any Γ arrival sequence can be mirrored by a Γ^{aux} arrival sequence, consisting of task instances with the same arrival times, priorities and execution times. Thus, if Γ^{aux} is schedulable under FPS, so is Γ. □

To analyse the auxiliary task set, we must allow deadlines to be larger than periods, as described above, but we also need to take into

consideration that tasks do not have unique priorities. In principle, we modify the analysis of the qth instance of t_i so that the interference from equally priorities tasks released before this instance is taken into account in the same way as the interference from previous instances of t_i.

We denote by $hp(i)$ and $ep(i)$ the tasks of higher priority than t_i, and tasks of equal priority, respectively. Note that $i \in ep(i)$ holds for any task t_i.

Definition 6.5. *Let* $hp(i) = \{j \mid P_j > P_i\}$ *and* $ep(i) = \{j \mid P_j = P_i\}$.

Next, we define $w_i(q)$, which denotes the latest possible finishing time, relative to the start of the busy period, of the qth instance of t_i, i.e., the instance arriving at time qT_i.

Definition 6.6.

$$w_i(q) = \sum_{\forall j \in ep(i)} \left(\left\lfloor \frac{qT_i}{T_j} \right\rfloor + 1 \right) C_j + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i(q)}{T_j} \right\rceil C_j$$

Note that, when t_i has a unique priority, the first sum in the definition above becomes $(q+1)C_i$, in which case the definition $w_i(q)$ is the same as in the original formulation by Tindell et al. where priorities are unique [147].

Finally, the worst case response time of t_i is established as follows:

Definition 6.7.

$$r_i = \max_{q=0,1,\ldots,\left\lfloor \frac{L(P_i)}{T_i} \right\rfloor} (w_i(q) - qT_i)$$

Proposition 6.3. *The original task set* Γ *is schedulable if* $r_i \leq D_i$ *for all tasks in* Γ^{aux}.

Proof. The qth instance of task t_i in the busy period is affected by two types of interference: (1) computations at higher priority that are released before the instance finishes, and (2) computations at the same priority that are released before the instance is released, or at the same time.

The first category is accounted for by the summation over $hp(i)$ in the definition of $w_i(q)$, following Tindell et al. [147]. The second category, together with the impact from the WCET of the instance itself, is accounted for by the summation over $ep(i)$ in the first term of the

$w_i(q)$ definition. The qth instance of task t_i is released at time qT_i, and by that time $\lfloor (qT_i)/T_j \rfloor + 1$ instances of t_j has been released. Any task instance with equal priority relesed after this time will not affect the response time, according to the FIFO assumption.

If $r_i \leq D_i$, then all instances of t_i in the busy period meet their deadlines, and thus Γ^{aux} is schedulable. According to Lemma 6.2, this implies that Γ is schedulable. □

Since $w_i(q)$ appears on both sides of the equation, it can not be computed directly. Instead, a sequence of values, $w_i^0(q),\ w_i^1(q),\ \ldots$, is generated, each of which is used in the right hand side of the equation when the next value in the sequence is computed. The process starts with the initial value $w_i^0(q) = 0$, and ends when a fixed point is found, or when a value greater than $qT_i + D_i$ is encountered, which indicates that the response time exceeds the deadline.

$$\begin{aligned} w_i^0(q) &= 0 \\ w_i^{n+1}(q) &= \sum_{\forall j \in ep(i)} \left(\left\lfloor \frac{qT_i}{T_j} \right\rfloor + 1 \right) C_j + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j \end{aligned}$$

The summation over equally prioritised tasks is the same for all values in the sequence. Also, the summation over higher priority tasks always becomes 0 when $w_i^1(q)$ is computed in the first step, since $w_i^0(q) = 0$. Thus, the equations can be reformulated as follows:

$$\begin{aligned} w_i^1(q) &= \sum_{\forall j \in ep(i)} \left(\left\lfloor \frac{qT_i}{T_j} \right\rfloor + 1 \right) C_j \\ w_i^{n+1}(q) &= w_i^1(q) + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j \end{aligned}$$

Example 6.3. For the example task set, we assume that T1 has highest priority, followed by P2 and finally T3. These priorities propagate to the auxiliary task set, as shown in Table 6.3.

The table also presents the response time for each tasks, computed according to the definitions above. Since $r_i \leq D_i$ for all tasks, the task set is schedulable. For a full explanation of how these values are derived, see Appendix C. ◇

Table 6.3: The auxiliary task set from Example 6.3, including response times.

From task	i	P_i	C_i	T_i	D_i	r_i
T1	1	High	10	50	30	10
P2	2	Mid	5	60	100	75
	3	Mid	25	70	100	75
	4	Mid	25	200	100	75
T3	5	Low	30	200	200	190

6.3 Scheduling with dynamic priorities

As an alternative to assigning a priority to each task statically, priorities can be based on task parameters that change dynamically, either from one instance of a task to another, or continuously during execution, e.g., the absolute deadline or remaining execution time. This requires more support from the runtime platform, and possibly introduces some additional overhead, but it can also allow a more efficient utilisation of computational resources.

The most common dynamic priority policy is *earliest deadline first* (EDF), in which the scheduler always selects a task for execution that have the earliest absolute deadline from those that are currently active. As shown by Dertouzos [45], EDF is optimal for preemptive, independent tasks, meaning that if a task set is not schedulable by EDF, then it can not be scheduled by any other scheduling policy either.

Chetto et al. [37] describe how EDF can be used for task sets with precedence constraints by modifying release times and deadlines, and Spuri [141] investigates the shedulability of tasks with shared resources and release jitter. Buttazzo and Stankovic [31] augment EDF with an online guarantee method to improve the behaviour of in overload situations, and Anderson et al. [13] use EDF for scheduling of multiprocessor soft real-time systems.

6.3.1 Pattern triggered tasks under EDF

The realisation of pattern triggered task, described in Section 6.1.2, can be used directly in an EDF scheduled system. When several instances of a pattern triggered task are active at the same time, the fact that their absolute deadlines follow the order in which they arrived guarantees that primitive events are processed in the correct order.

For the schedulability anlaysis, we fist prove the following lemma:

Lemma 6.4. *If Γ^{aux} is schedulable under EDF, then Γ is schedulable under EDF.*

Proof. Following the proof of Lemma 6.2, we can show that any Γ arrival sequence can be mirrored by a Γ^{aux} arrival sequence consisting of task instances with the same arrival times, deadlines and execution times. Thus, if Γ^{aux} is schedulable under EDF, so is Γ. □

Under EDF, the auxiliary task set can be analysed by existing techniques without modification. We outline the principles below, and refer to the literature for details and proofs [30, 63].

Definition 6.8. *The processor utilisation U of a task set $\{t_1 \ldots, t_n\}$ is defined by*

$$U = \sum_{i=1}^{n} C_i/T_i$$

Under the restriction that $D_i = T_i$, the processor utilisation provides a sufficient and necessary condition for schedulability [93], but when deadlines and periods are unrelated, it only gives a necessary condition.

Proposition 6.5. *If $U > 1$, then the task set is not schedulable.*

Example 6.4. For the auxiliary task set in Example 6.2, we have the following processor utilisation:

$$U = (10/50) + (5/60) + (25/70) + (25/200) + (30/200) \approx 0.915$$

Since $U \leq 1$, this does not help determine whether the auxiliary task set is schedulable or not. However, if we for example want to investigate if it would be possible to double the frequency of the task with lowest priority, i.e., to change the value of T_4 from 200 to 100, we get

$$U = (10/50) + (5/60) + (25/70) + (25/200) + (30/100) \approx 1.07$$

which directly indicates that the auxiliary task set is not schedulable by any scheduling policy if this change is made. ◇

If $U \leq 1$, a more detailed analysis is required, in which deadlines are taken into consideration. As shown by Liu and Layland [93] for task sets where $D_i = T_i$, and extended to less restricted task sets by Spuri [141] and Ripoll et al. [128] independently, a given task set is schedulable under EDF if and only if no deadline is violated during the busy period.

Thus, assume that all tasks are released at time 0, and that subsequent instances are released according to the period or the minimum interarrival time. To ensure schedulability, we must check for each absolute deadline d within the following busy period that the total amount of computation required to finished before d can be served within the interval $[0, d]$.

Definition 6.9. *Let* $\Gamma^{\text{aux}} = \{t_1 \ldots, t_n\}$*, with busy period* L*, and define*

$$\mathcal{D} = \{d \mid d = qT_i + D_i,\ d \leq L,\ 0 \leq i \leq n,\ q \geq 0\}$$

Definition 6.10. *The processor demand in the first* d *time units of the busy period, denoted by* $h(d)$*, is defined as follows:*

$$h(d) = \sum_{D_i \leq d} \left(1 + \left\lfloor \frac{d - D_i}{T_i} \right\rfloor\right) C_i$$

Proposition 6.6. *The original task set* Γ *is schedulable if* $h(d) \leq d$ *for all* $d \in \mathcal{D}$*.*

Proof. We provide only an informal line of argumentation, and refer to Baruah et al. [18] and Spuri [141] for details.

If a deadline violation can occur, it will manifest during the busy period following the simultaneous release of all tasks. So, consider the qth instance of t_i, with a deadline $d = qT_i + D_i$ within the busy period, i.e., $d \in \mathcal{D}$. In order for this instance to meet the deadline, all instances with release time and absolute deadline in the interval $[0, d]$, including the instance itself, must be finished by the time d. This amount of computation is represented by $h(d)$. Thus, since there is no idle processor time during the busy period, the instance will meet the deadline if and only if $h(d) \leq d$.

If $h(d) \leq d$ for each absolute deadline within the busy period, then Γ^{aux} is schedulable, and according to 6.4, this implies that Γ is schedulable. □

This schedulability test can be performed more efficiently by reducing the size of $\mathcal{D}$ in different ways that still guarantee that any overload situation will manifest at one of the remaining absolute deadlines [159, 129]. An overview of these techniques is given by George et al. [63].

Example 6.5. For the auxiliary task set of our example, shown in Table 6.2, we have a busy period $L = 190$, and the following absolute deadlines must be investigated:

$$\mathcal{D} = \{30,\ 80,\ 100,\ 130,\ 160,\ 170,\ 180\}$$

For each deadline in this set, we compute $h(d)$ and check that $h(d) \leq d$:

$$\begin{array}{lll} h(30) = 10 & \leq & 30 \\ h(80) = 20 & \leq & 80 \\ h(100) = 75 & \leq & 100 \\ h(130) = 85 & \leq & 130 \end{array} \qquad \begin{array}{lll} h(160) = 90 & \leq & 160 \\ h(170) = 115 & \leq & 170 \\ h(180) = 125 & \leq & 180 \end{array}$$

Since all the above inequalities hold, the task set is schedulable under EDF. This should not come as a surprise, since we previously showed that it is schedulable with fixed priorities, and thus the optimality of EDF ensures that it is schedulable under EDF as well.

A detailed description of how $\mathcal{D}$ and the $h(d)$ values are derived can be found in Appendix C. ◇

Chapter 7

Event Pattern Triggered Components

SaveCCM is a component model intended for development of embedded software for vehicular systems [6], developed in the SAVE project [134]. The targeted application domain makes resource efficiency a major concern, which means that the run-time framework governing e.g., component communication, must be particularly lightweight. Furthermore, system behaviour should be predictable, both functionally and with respect to timeliness and resource usage. An important characteristic of SaveCCM is that the transfer of control between a set of components is handled by explicit trigger ports. The way in which these ports are connected, determines the order in which components execute.

This chapter presents how the component triggering in SaveCCM can be extended by means of an event algebra, allowing components to be triggered by complex event patterns in addition to clock signals and individual external events. Separating the detection of triggering conditions from the definition of the triggered services permits more general components and thus improves component reusability. Providing event detection mechanisms within the component model means that triggering conditions are explicitly available for system analysis at design time. For example, more accurate timing information can be obtained compared to implicit pattern detection within the components, which can allow a higher utilisation of computational resources.

The extension has been integrated into an early SaveCCM tool pro-

totype by which a system description in XML format can be synthesised into code for the real-time operating system RTXC [126]. It is also possible to run the synthesised code in the simulation environment CC-SimTech [106], for the purpose of testing and debugging.

7.1 SaveCCM syntax and semantics

The graphical notation of SaveCCM, presented in Figure 7.1, is based on a modified subset of UML 2.0 component diagrams. The semantics is formally defined by a two-step transformation, first from the full language to a similar but simpler language called SaveCCM Core, and then into timed automata with tasks. For details on the formal semantics, see Carlson et al. [33].

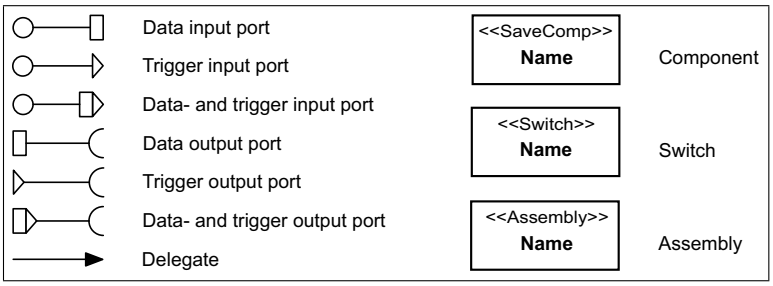

Figure 7.1: The graphical notation of SaveCCM.

In SaveCCM, systems are built from interconnected elements with well-defined interfaces consisting of input- and output ports. The three element categories; components, switches and assemblies, are described below. The model is based on the control flow (pipes-and-filters) paradigm, and an important feature is the distinction between data transfer and control flow. The former is captured by connections between *data ports* where data of a given type can be written and read, and the latter by *trigger ports* that control the activation of components. A port can also have both triggering and data functionality.

This separation of data and control flow allows components to exchange data without handing over the control, which simplifies communication between sub-systems running at different frequencies, and the

construction of feedback loops. Another aspect of explicit control flow is that the resulting design is sufficiently analysable with respect to temporal behaviour to allow analysis of schedulability, response time, etc., which is crucial to ensure correctness of real-time systems.

Components

Components are the main architectural element in SaveCCM. In addition to input and output ports, the interface of a component contains a series of quality attributes, such as (worst case) execution time information for a number of target hardware configurations, reliability estimates, safety models, etc. The quality attributes are used for analysis, model extraction and for synthesis.

To achieve predictability and to facilitate analysis, components are subject to a fairly restricted execution semantics. A component is initially inactive, and remains in this state until all input trigger ports have been activated, at which point it switches to the executing state. In a first phase of its execution, the component reads all input data ports. It then performs the associated computations on the basis of this input and possibly an internal state. When the computation phase is over, the output is written to the output data ports. Finally, the input trigger ports are reset and all outgoing trigger ports are activated, after which the component returns to the idle state.

This restricted "read-execute-write" semantics ensures that once a component is triggered, the execution is functionally independent of any concurrent activity. In particular, a component produces the same output with preemptive and non-preemptive scheduling, i.e., whether or not a task may be interrupted by another task during its execution. Hence, component execution can be abstracted by a single transfer function from input values and internal state to output values.

Switches

Switches provide the means to change the component interconnection structure, either statically for pre-runtime configuration, or dynamically, e.g., to implement modes and mode switches. The switch specifies a number of connection patterns, i.e., partial mappings from input to output ports. Each connection pattern is guarded by a logical expression over the data available at the input ports of the switch, defining the condition

under which that pattern is active.

Unlike components, switches are not triggered. Instead, they respond directly to the arrival of data or a trigger signal at an input port and immediately relay it according to the currently active connection patterns. Switches perform no computation other than the evaluation of connection pattern guards.

Assemblies

Assemblies are encapsulated sub-systems. The internal components and interconnections are hidden from the rest of the system, and can be accessed only indirectly through the ports of the assembly. Like switches, assemblies are not triggered. Data and trigger signals arriving at a port of an assembly are immediately relayed via the outgoing connections.

Ports and Connections

Component input ports are one-place buffers with overwrite semantics, as are switch input ports that occur in some connection pattern guard. Other ports, in particular component output ports and assembly ports, are just conceptual interaction points through which data passes.

Most connections are *immediate*, meaning that they represent lossless, atomic migration of data or trigger signals from one port to another, but there are also *complex* connections for which characteristics such as delay, information loss, etc., can be specified by timed automata.

Following UML 2.0, a connection from an assembly input port to an input port of an internal element, or from an internal output port to an assembly output port, is denoted by a delegation arrow, but semantically they are equivalent to ordinary connections from output to input ports.

7.2 Event pattern triggering

On system level, the trigger port connections describe which of the components that execute in response to a certain external event or periodic activation, and restrict the order in which they may be executed. For an individual component, however, the triggering signal can be seen as an event to which the component should respond by executing. From this perspective, the SaveCCM semantics only support the publishing

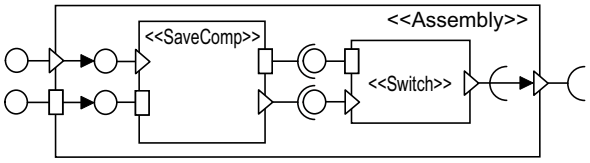

Figure 7.2: The triggering of this alarm assembly is forwarded to the output trigger port only when the input data is outside a given range.

of a single event from each component, representing that the component has finished its execution. This event is automatically published by the underlying framework, and can not be explicitly managed by the component.

Nevertheless, since switches allow dynamic rewiring of trigger port connections based on values produced by the component, we can construct assemblies that have some control over the output trigger ports. The assembly is still inactive until triggered, and thus can not freely activate the output trigger port at any time, but once triggered, it can decide for each of the output trigger ports if the triggering should be forwarded or not. For example, Figure 7.2 shows the design of an alarm assembly with optional outgoing triggering. When triggered, the component checks if the input is outside the range of acceptable values and writes true or false to the output port, accordingly. The switch contains a single connection from input to output, guarded by the boolean value at the data port. Thus, the triggering signal is only forwarded from the assembly if the value at the input data port is outside the range. To the rest of the system, connecting to this trigger port corresponds to subscribing to an alarm event.

7.2.1 Event elements

To incorporate event pattern detection into the component model, the SaveCCM notation is extended with a new element, termed *Event*. An event element has a number of named input trigger ports with or without data capabilities. The input port names act as primitive events in a collection of event expressions that specify the event patterns of interest, and each expression is associated with an output trigger port which is activated when an occurrence of that pattern is detected.

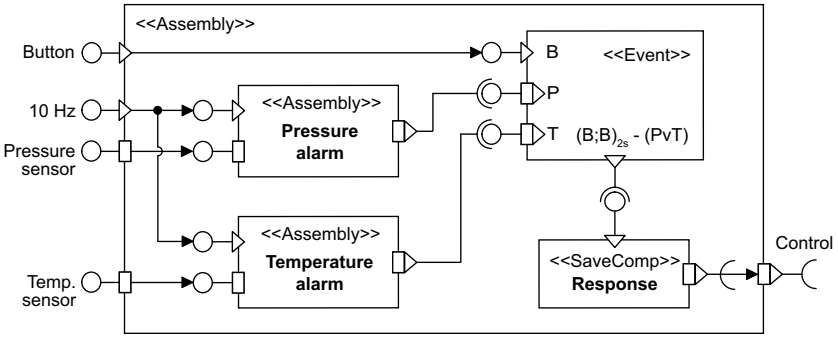

Figure 7.3: An example of the extended SaveCCM syntax.

Example 7.1. Figure 7.3 shows an example of a system where the event element is used. Continuing the example from previous chapters, the desired behaviour is to carry out a particular response when the button is pressed twice within two seconds, unless either of the alarms occurs in between.

The two alarm assemblies read sensor values at a fixed frequency, and signal an alarm via the output port if conditions are unsatisfactory. The event element specifies the situation we are interested in, and when this situation is detected, the appropriate response is carried out by the designated component. ◇

The semantics of the event element is defined by a transformation into standard SaveCCM constructs. The major part consists of one component for each event expression, where the pattern detection is performed. Code for these components is automatically generated from the event expressions, following the detection algorithm presented in Section 5.1. In addition, a number of auxiliary components are needed to tweak the standard SaveCCM triggering semantics, and switches to control whether the triggering should be forwarded to the output ports or not.

Figure 7.4 illustrates this transformation by showing the semantic equivalence of the event element in Figure 7.3. Formally, the semantics is specified by Definition 7.1 below.

Definition 7.1. *The semantics of an event element with input ports* $p_1, \ldots, p_n$*, and output ports* $e_1, \ldots, e_m$ *associated with event expressions* $E_1, \ldots, E_m$*, is defined to be the same as that of an assembly with the same input and output ports, and the following internal structure:*

- *For each* $1 \leq i \leq n$*, the assembly contains a component* A_i *with one input trigger port* a_i *and an output trigger port* b_i*. In the case when* p_i *is a combined data and trigger port,* A_i *also has a data output port* c_i*. When triggered,* A_i *copies the data from* a_i *to* c_i*, and writes the number* i *and a timestamp to* b_i*.*

- *For each* $1 \leq j \leq m$*, the assembly contains a component* D_j *with one input trigger port* f_j *and a data input port* d_{ji} *for each* i *such that the name of* p_i *is in* $\mathrm{prim}(E_j)$*.* D_j *has a single output data output port* g_j *and a trigger output port* h_j*. When* D_j *is triggered, it examines the value of* f_j *to know which of the primitive events the triggering corresponds to. It executes the detection algorithm for* E_j *once. If an occurrence of* E_j *is detected, the associated value is written to* h_j *and* true *is written to* g_i*. If no occurrence is detected,* false *is written to* g_i*.*

- *For each* $1 \leq j \leq m$*, the assembly contains a switch* S_j *with an input trigger port* k_j*, an input data port* s_j*, an output trigger port* o_j*, and a single connection pattern* $k_j \rightarrow o_j$ *guarded by* s_j*.*

- *For each* $1 \leq i \leq n$ *and* $1 \leq j \leq m$*, the assembly contains the following internal connections:*

$p_i \rightarrow a_i$	(delegation)		$g_j \rightarrow s_j$		
$b_i \rightarrow f_j$	if d_{ji} exists		$h_j \rightarrow k_j$		
$c_i \rightarrow d_{ji}$	if d_{ji} exists		$o_j \rightarrow e_j$		(delegation)

Also, port types are assigned as follows:

$$\begin{aligned}
\mathrm{type}(a_i) &= \mathrm{type}(c_i) = \mathrm{type}(d_{ji}) = \mathrm{type}(p_i) \\
\mathrm{type}(b_i) &= \mathrm{type}(f_j) = \mathrm{int} \\
\mathrm{type}(g_j) &= \mathrm{type}(s_j) = \mathrm{bool} \\
\mathrm{type}(h_j) &= \mathrm{type}(k_j) = \mathrm{type}(o_j) = \mathrm{type}(e_j)
\end{aligned}$$

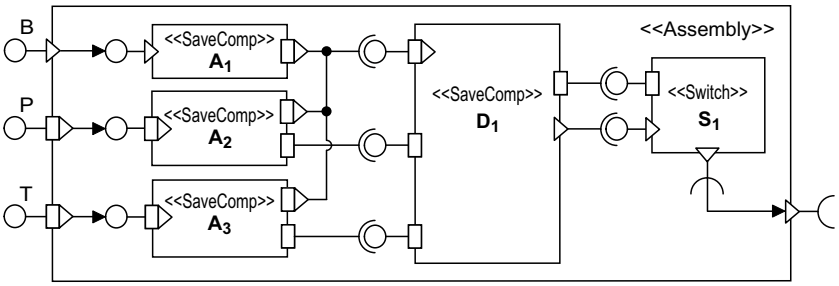

Figure 7.4: Translation of the event element in Figure 7.3 into ordinary SaveCCM constructs.

To ensure that all primitive event occurrences are processed by the detection mechanism, it must be guaranteed that a triggered component never receives a new triggering signal before it becomes idle again. This issue is not specific to the event pattern extension, and Åkerholm et al. [6] describe how TIMES [12], a design and analysis tool for real-time embedded systems based on the model checking tool UPPAAL [88], can be used to verify that all triggerings are preserved.

7.2.2 Synthesis

The synthesis of executable software from a SaveCCM system description is performed in four steps: task allocation, attribute assignment, code generation and compilation [7]. In the task allocation phase, triggering connections are explored to identify components that can be fixed in a static execution order without compromising the triggering semantics of SaveCCM. Allocating such components to the same task can improve performance, since data transfer between components in the same task can be efficiently implemented by shared variables, and triggering is reduced to procedure calls. For components in different tasks, communication is done via persistent message channels handled by the runtime framework. Once the allocation of components to tasks is done, task attributes can be derived from component attributes and connection information. Task code is generated, consisting of calls to component code, evaluation of switch guards and additional glue code for communication within and between tasks. Finally, the code is compiled for the particular

target hardware.

Since the new event element is defined by a transformation into standard SaveCCM concepts, no changes have to be made to the synthesis mechanism. In the current prototype, however, specialised code is generated for event elements to avoid some of the communication overhead introduced by the transformation. The event detection code is generated based on the algorithm presented in Section 4.3, optimised for static event expressions.

7.2.3 Analysis support

One argument for extending the component model with event pattern detection is that it facilitates analysis on a system design level, compared to developing a new ad hoc component for the detection of a particular triggering condition. Defining the event element in terms of standard SaveCCM constructs means that the standard analysis techniques can still be applied, including the triggering preservation property discussed above, and application specific liveness and safety properties.

In addition to properties that can be investigated by standard analysis methods, some properties of the event element depend on the event expressions, and thus require special treatment.

Memory and worst case execution time

The memory footprint and execution time of an event element can be established by analysis of the generated code, using standard techniques. Alternatively, analysis can be based directly on the event expression and the types of the input ports, as described in Section 5.5. This alternative give concrete results for memory footprint, since the underlying representation of primitive types is known, but worst case execution time is given in an abstract form, expressed in terms of the number of assignments, comparisons, arithmetical operations, etc. that are executed in the worst case.

Triggering frequencies

Straightforward triggering analysis can establish that an output triggering port with expression E_j is potentially activated in response to the activation of any individual input triggering port in $\text{prim}(E_j)$, but this is clearly an overapproximation for many expressions. Using the idea from

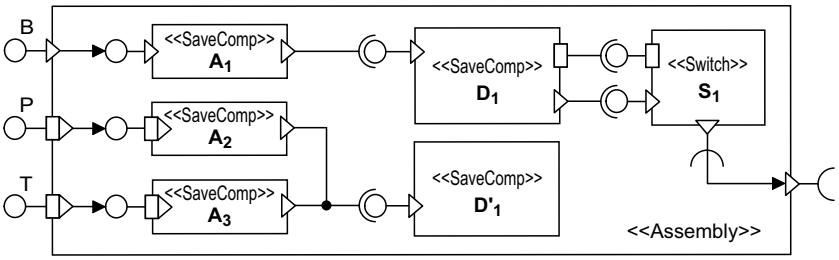

Figure 7.5: Alternative translation of the event element in Figure 7.3, used for temporal analysis.

Chapter 6, we note that only activations of triggering ports in term(E_i) can in fact result in an activation of the E_i port.

In order to take this into account, a different transformation can be used to define event elements for the sake of temporal analysis, i.e., for analyses techniques that do not depend on the functional behaviour of components, only their execution time. This transformation includes an additional copy of each D_j component, with the same WCET attribute but with no output ports. An input port p_i is connected to d_{ij} at the original D_j component only if the name of p_i is in term(E_j), otherwise it is connected to the new D_j component with no output ports. Thus, activations of ports that are in prim(E_j) but not in term(E_j) result in the execution of the detection mechanism, but the triggering is not forwarded.

Example 7.2. We consider again the event element from Example 7.1. Figure 7.5 shows the result of the alternative transformation for temporal analysis purposes. Since term$((\mathsf{B};\mathsf{B})_{2s} - ((\mathsf{P} \vee \mathsf{T})) = \{\mathsf{B}\}$, the auxiliary components for P and T (i.e., A_2 and A_3) are connected to the copy of D_1 without output trigger ports.

When the whole system (shown in Figure 7.3) is considered, the analysis can determine that the response component is never triggered by the periodic 10 Hz clock. If event pattern detection had been performed by user defined component code, this would have been more difficult to discover. ◇

Chapter 8

Conclusions

This chapter summarises the contributions of the thesis, and describes how they relate to existing work in the area. It also presents a number of directions in which this work could be extended in the future.

8.1 Summary and contributions

Many computer systems are to some degree influenced by occurrences of external or internal events. In some applications, individual event occurrences are not as important as certain patterns of event occurrences representing specific situations that the system should respond to. Having a dedicated mechanism for detecting event patterns that are of interest to the system, instead of performing implicit detection locally, means that the patterns are explicitly available for analysis and optimisation. We have addressed event pattern detection in the context of embedded real-time systems, characterised by limited memory and processing resources and by the need for predictable temporal behaviour.

The thesis presents an event algebra consisting of five operators, by which the simple events of a system can be combined to express complex event patterns. The algebra has a simple and intuitive semantics, and any pattern that can be expressed by the algebra can be efficiently detected with limited resources in terms of time and memory. The thesis also presents schedulability theory for systems consisting of both periodic activities and activities triggered by event patterns, and shows how the event algebra can be used within a component model for embedded

systems. Below, the summary of contributions given in the introduction (on page 5), is repeated in a more elaborate form.

- The event algebra provides overlapping detection of event patterns, meaning that pattern occurrences that overlap in time can be detected. It supports event parameters, allows simultaneous event occurrences, and can express non-occurrence of complex patterns.

  The algebra semantics is interval based, meaning that pattern occurrences have duration. This is an important property, since it permits an operator semantics that complies with the expected, intuitive meaning of concepts such as "*followed by*", to a greater extent than semantics based on single time points would [55]. The somewhat presumptuous claim that the algebra operators behave according to intuition is backed up and formalised by a number of algebraic laws that are satisfied by the algebra.

  In order to deal with resource constraints, a formal restriction policy is applied, specifying that each time there is one or more occurrences of the pattern, one of these occurrences must be detected.

- An imperative detection algorithm is presented for which bounds on memory footprint and execution time can be statically determined for any pattern defined by the event algebra. The resource bound does not depend on the time values used in the specification of the pattern or the frequency of primitive event occurrences. A complexity analysis and experiments on randomly generated patterns indicate that the algorithm is sufficiently efficient for the target domain.

- The notion of *pattern triggered tasks* is introduced, meaning tasks that are triggered by occurrences of certain event patterns. This concept facilitates the design of embedded systems with activities that should be performed in response to event patterns, by making pattern definitions explicit and thus available for analysis.

  We show how pattern triggered tasks can be realised without support from the underlying operating system, by means of ordinary event triggered tasks that perform the pattern detection as a first step of their execution.

  Two schedulability analysis techniques are presented, for fixed priority scheduling and for scheduling under EDF, respectively, by

which a mixed task set consisting of both periodic and pattern triggered tasks can be analysed to determine if deadlines are met under all circumstances. The analysis uses information from the pattern definitions to achieve a more accurate result than what would be possible if pattern detection was performed implicitly within the user defined task code, and this improved accuracy allows better utilisation of processor resources.

- The thesis also shows how the event algebra can be used to extend the component triggering mechanism of SaveCCM, a component model for embedded vehicular systems. By performing event pattern recognition as a separate activity, components can be made more general, which improves component reusability. Providing detection as a part of the underlying framework, instead of letting the user implement it when needed, means that triggering conditions are explicitly available for system analysis at design time, which can improve the accuracy when analysing temporal system properties.

8.2 Comparison with related work

This section relates the contributions described above to related work in the area. Detailed descriptions of the frameworks and languages discussed here are presented in Chapter 3.

8.2.1 Active databases

The proposed algebra was originally influenced by work in the area of active databases, in particular the event specification language Snoop [35], which is visible in the choice of operators. Also, the restriction policy used to achieve bounded-resource detection resembles the concept of *parameter contexts* from this domain. An important distinction, however, is that parameter contexts are applied to the individual operators of an expression, while our restriction policy is applied once to the expression as a whole, which we believe makes it easier to understand its impact on the algebra operators. On a more concrete level, the recent context, as it is defined for interval based semantics [3, 100], selects the constituent event with maximum end time, while the restriction policy compares start times.

None of the work we have considered from this domain present algebraic laws for their operators, and most of them have semantics based on single time points which makes it difficult to achieve the expected behaviour for some operator combinations [55]. Approaches with interval based semantics include Solicitor [100], the work of Roncancio [130], and the interval semantics for a subset of Snoop developed by Adaikkalavan and Chakravarthy [3].

The use of interval semantics in our algebra also enables a binary negation operator, representing non-occurrence of one pattern during an occurrence of another pattern. This operator is more general than the ternary negation found in, e.g., Snoop [35], SAMOS [57] and the metamodel developed by Zimmer and Unland [161], where the non-occurrence interval must be specified by two patterns that mark the start and end of the interval. SAMOS [57] and Ode [59], are also different from our work in that they do not allow simultaneous occurrences of primitive events.

Unlike our work, resource usage is not a main focus of these methods, with a few exceptions. In Snoop, all operators can be implemented with bounded memory in the recent context [35], but not in the other contexts. Solicitor permits bounded-memory detection in all contexts for patterns that have explicit limitations on occurrence durations [100]. However, the amount of memory required to detect a pattern depends on the time values used within the pattern specification and the minimum interarrival times of primitive events, while the resource demands of our algebra depends on the size of the expression only.

8.2.2 Automata and regular expressions

For event pattern detection mechanisms based on finite automata or regular expressions, resource usage is bounded by default. The general drawbacks of these approaches compared to our method are the difficulty of overlapping detection, and the complexity associated with specifying non-occurrence of complex patterns and other constructs that require concurrent detection of subpatterns [109]. Also, simultaneous occurrences are not easily supported, as discussed in Section 3.2.

ECL and PAR [131] are similar to regular expressions in style, and the detection can always be performed with bounded memory. Contrasting our algebra, they are procedural in style and support single or repeated detection, but not overlapping. Moreover, simultaneous primitive occurrences are not supported, and patterns can only refer to the

order of constituent occurrences, not to the time between occurrences.

The composite event detection automata framework developed by Pietzuch [123], focuses on distributed systems where it is not always possible to determine the correct ordering of primitive event occurrences, which is not addressed by our algebra. Also, some concepts in the framework lack formal definitions, which prevents a detailed evaluation.

8.2.3 Temporal logic

When temporal logic is used for event pattern detection, not only to define the semantics but as basis for the concrete detection mechanism as well, the high expressiveness typically means that not all patterns can be detected with bounded resources. For example, in the FTL and PTL approach by Sistla and Wolfson [139], the amount of stored information increases over time [38]. The work by Chomicki on past first-order temporal logic permits detection with bounded memory, but the size is bounded by the size of the value domains used in the database [38]. Real time logic also performs detection with limited memory, but the memory footprint for a particular pattern depends on the time values used within the pattern specification, and on the occurrence frequency of primitive events [103]. EAGLE [17], as well as the FTL/PTL framework [139], does not support detection of partially overlapping occurrences, only single or repeated detection is addressed.

8.2.4 Additional work on event notification

Chronicle recognition is able to detect overlapping occurrences, but this comes at the cost that no memory bound can be given in the general case, only under the restriction that all patterns have duration bounds [47]. In practice, chronicle recognition has proven sufficiently efficient for industrial real-time systems, e.g., online monitoring of gas turbines [5].

The event notification service READY [65], and the event algebra developed by Hinze and Voisard [75] do not address resource efficency or bounded detection. The composite event detection in ECCO [157] is based on a combined approach of our algebra [157] and the work of Pietzuch et al. [123], but it unclear what, if any, properties of our algebra that hold also in their work, since no formal semantics is presented. We have found no work on composite event notification services that provide algebraic laws for their event algebra operators.

8.2.5 Real-time scheduling

Schedulability analysis of tasks that are triggered by events with non-trivial arrival patterns has been addressed in numerous ways (see for example Buttazzo [30] or Klein et al. [83]). The simplest approach is to safely approximate such tasks by periodic ones. If a strictly periodic approximation is not sufficiently precise, more elaborate models can be used, for example sporadic activation bursts where each burst consists of a bounded number of triggerings that can be arbitrarily close in time [83]. There are also approaches that allow more detailed models of how triggering events can arrive, for example the analysis tool TIMES, where the triggering of tasks can be modelled by timed automata [12].

From the point of view of the system, however, such tasks are still triggered by occurrences of a single event, even though the occurrences of this event may be modelled in some way based on information about the environment. The concept of pattern triggered tasks, presented in this thesis, is different in that it includes event pattern detection as part of the design of a real-time system.

The work by Berndtsson and Hansson concerns databases that combine active functionality in the form of ECA rules, with real-time constraints such as bounded response times [21]. They address how event detection activities can be mapped onto tasks, and propose that events are processed based on their respective criticality rather than in the order of occurrence, to overcome problems such as event bursts and transient overload. The realisation of pattern triggered tasks in our approach is based on the same observation that detection activities should be processed based on the criticality of the response. Their work differ from ours in that they focus on database applications, and the use of ECA rules means that they have to address additional issues concerning the condition part. Also, they assign priority levels to individual events, both primitive and composite, while we assign priority to whole patterns.

8.2.6 Component models

For general purpose component models, advanced event handling such as pattern detection can be introduced as additional middleware functionality, as in COBEA [97] which, among other things, provide composite event detection for the CORBA Event Service [113]. Component models for small embedded systems require lightweight runtime frame-

works [107], and component deployment and composition are preferably performed at design time to achieve better performance and more predictabile behaviour and to simplify verification [43, 44]. This suggests that a more static approach to event pattern detection is preferable over advanced, dynamic middleware services. In our proposed extension of SaveCCM, pattern specifications are formulated at design time and turned into concrete detection mechanisms for those particular patterns during synthesis, which facilitates static analysis of, e.g., resource usage and temporal properties.

8.3 Future work

A number of future research directions have been identified. Some of them concern the algebra as such, and others address the larger perspective of how this algebra can be used as a part of a larger system.

8.3.1 Non-instantaneous primitive events

One assumption in the current version of the algebra is that all primitive event occurrences are instantaneous, but the algebra semantics would not be affected if this restriction was removed. In order to still ensure bounded memory detection for arbitrary expressions, however, the primitive events would have to be restricted somehow to ensure that the information about "possible future start times", which is propagated between subexpressions in the detection algorithm, is still available in some form.

Informally, a non-instantaneous primitive event can be handled if the event source notifies the detection mechanism both when an occurrence starts and when it ends, and if the number of simultaneously active "possible start times" is bounded, for example if start and end notifications are strictly alternating. As future work, this could be investigated further to formulate exact criteria under which non-instantaneous primitive events can be allowed.

8.3.2 Expressiveness

The algebra operators were selected because they, or related constructs, occur in many other techniques for event pattern specification, but it remains on open issue how well these operators meet the requirements

of different application domains. To remedy this, it would be good to investigate the expressiveness of the current algebra, from a theoretical point of view, but also more pragmatically by performing case studies in the targeted domain.

As a result of the interval based semantics, there are many ways in which two occurrences can be related compared with single point semantics (see Figure 2.7 on page 25). This means that there are several "potential operators" between sequence, where the order of the constituent occurrences is fixed and overlapping is not allowed, to conjunction, with no restriction on the constituent occurrences. There are also operators in other methods, such as the simultaneous conjunction in Ode [59] or the aperiodic construct in Snoop (described in Section 3.3.3), that could possibly be incorporated in some form in our algebra.

The temporal restriction can be used to specify that some events occur within a given length of time. However, the current algebra does not support specification of timeouts, e.g., an occurrence of A *not* followed by B within τ time units. One way to allow this is to add a delay operation (here denoted by the symbol $\triangleright$) similar to the one used by Mellin [100]. Informally, for any occurrence of A there is an occurrence of $A \triangleright \tau$ with the same start time, but ending τ time units later then the end time of the A occurrence. Then, the timeout event in the example above could be defined as $(A \triangleright \tau) - B$. It should be possible to implement this operator with bounded memory, and also to relay the information about possible start times, which is required by the algorithm. However, with this extension, the memory footprint of the detection mechanism would no longer be bounded by the expression size, since it would also depend on the timeout lengths and on event occurrence frequencies.

8.3.3 Optimisation and more detailed WCET

Section 5.1 describes how the detection algorithm can be optimised when the expression is static and known at compile time. In addition to this, it is possible to optimise detection with respect to what primitive events that occurred in the current tick. In particular, when the algorithm is executed once for each primitive occurrence, rather than once each tick, it can be very efficient to have dedicated variants of the algorithm for different primitive events. Persistent variables are shared among the algorithm variants, and each variant is optimised based on the knowledge of which primitive that occurred.

A consequence of this optimisation is that the execution time of the detection algorithm can differ significantly depending on what primitive event occurrence that triggered the detection. This information could be captured by replacing the single WCET value $\text{wcet}(E)$ with one value for each primitive event, i.e. $\text{wcet}(E, A)$ defined for each $A \in \text{prim}(E)$. It should be possible to modify the analysis described in Section 5.5 to produce this information. To take advantage of this information in the schedulability analysis, $\text{wcet}(E_i, A)$ can be used instead of $\text{wcet}(E_i)$ in the definition of parameters for the tasks in the auxiliary task set.

In addition to staticly optimising the algorithm by identifying parts of the expression that can be ignored when detection is triggered by a particular primitive event, it is also possible to do something similar dynamically, based on the successful detection of subpatterns. Rather than processing all parts of the tree that are potentially affected by the current occurrence, the bottom-up traversal can stop when a subexpression is processed that does not result in a full detection. Although this would not improve the worst case performance, it might have a significant impact on the average performance.

Further optimisations include investigating if certain operator combinations can be given dedicated, optimised implementations, and to experiment with optimising expression transformations based on the algebraic laws, which was done for an earlier version on the algebra [32].

8.3.4 Specification of triggering patterns

The idea of pattern triggered tasks could be further investigated outside the context of this event algebra. For example, one could consider other ways of specifying triggering patterns, and compare the usability of different approaches. The schedulability analysis is based on the fact that the triggering pattern can be approximated by a disjunction of a subset of the primitive events that occur in the pattern, which allows the task to be approximated by a collection of sporadic tasks. This idea could be reused for other techniques as well, but there might also be other analysis approaches that give more precise approximations.

8.3.5 Optional triggering in SaveCCM

The triggering semantics in SaveCCM specifies that a component always activates its output trigger ports after execution. When triggering sig-

nals are seen as events, this corresponds to allowing just a single event to be published from each component, and this event is controlled by the underlying framework. As described in Section 7.2, switches can be used to construct assemblies that control which of its output trigger ports to activate, which strengthens the analogy between triggering and events.

As future work, the SaveCCM language could be extended so that components can have additional output trigger ports that are not controlled by the framework. This would allow SaveCCM to be used in a more event oriented style, which is interesting to us since it increases the usefulness of pattern detection, but it might also add strength to SaveCCM and possibly broaden the application domain. The main question is whether this change would have a negative impact on the desired properties of SaveCCM, but if mandatory and optional trigger ports are distinguished syntactically, it should still be possible to achieve the same level of predictability for applications where this is required.

Appendix A

Proofs

Theorem 4.1 If $res(S, S')$ and $res(T, T')$ hold, than for any event stream U and $\tau \in \mathcal{T}$ the following implications hold:

- $res(\mathrm{dis}(S', T'),\ U) \quad \Rightarrow \quad res(\mathrm{dis}(S, T),\ U)$
- $res(\mathrm{con}(S', T'),\ U) \quad \Rightarrow \quad res(\mathrm{con}(S, T),\ U)$
- $res(\mathrm{neg}(S', T'),\ U) \quad \Rightarrow \quad res(\mathrm{neg}(S, T),\ U)$
- $res(\mathrm{seq}(S', T'),\ U) \quad \Rightarrow \quad res(\mathrm{seq}(S, T),\ U)$
- $res(\mathrm{tim}(S', \tau),\ U) \quad \Rightarrow \quad res(\mathrm{tim}(S, \tau),\ U)$

Proof. We prove each implication in a separate case:

Disjunction case: Assume $res(\mathrm{dis}(S', T'), U)$. Then, for any $u \in U$ we have $u \in \mathrm{dis}(S', T')$ and thus $u \in S' \cup T'$. Since $S' \subseteq S$ and $T' \subseteq T$, we have $u \in S \cup T$, implying $u \in \mathrm{dis}(S, T)$. Thus $U \subseteq \mathrm{dis}(S, T)$, which satisfies the first constraint in the definition of *res*.

Next, take an arbitrary $u \in \mathrm{dis}(S, T)$. Then $u \in S \cup T$ and according to the definition of *res* there must exist an $u' \in S' \cup T'$ such that $\mathrm{start}(u) \leq \mathrm{start}(u')$ and $\mathrm{end}(u') = \mathrm{end}(u)$. We have $u' \in \mathrm{dis}(S', T')$ and thus $res(\mathrm{dis}(S', T'), U)$ implies that there exists an $u'' \in U$ with $\mathrm{start}(u') \leq \mathrm{start}(u'')$ and $\mathrm{end}(u'') = \mathrm{end}(u')$. Since this means that $\mathrm{start}(u) \leq \mathrm{start}(u'')$ and $\mathrm{end}(u'') = \mathrm{end}(u)$, the second constraint in the definition of *res* is satisfied.

Finally, $res(\mathrm{dis}(S', T'), U)$ ensures that all instances in U have different end times. Together, this gives $res(\mathrm{dis}(S, T), U)$.

Conjunction case: Assume $res(\text{con}(S', T'), U)$. Then, for any $u \in U$ we have $u \in \text{con}(S', T')$ and thus $u = s \oplus t$ with $s \in S'$ and $t \in T'$. By the subset requirement in the definition of *res*, $s \in S$ and $t \in T$. So $u \in \text{con}(S, T)$ and thus $U \subseteq \text{con}(S, T)$.

Next, take an arbitrary $u \in \text{con}(S, T)$. Then $u = s \oplus t$ with $s \in S$ and $t \in T$, and by the definition of *res* there exists $s' \in S'$ and $t' \in T'$ with $\text{start}(s) \leq \text{start}(s')$, $\text{end}(s') = \text{end}(s)$, $\text{start}(t) \leq \text{start}(t')$ and $\text{end}(t') = \text{end}(t)$. Let $u' = s' \oplus t'$. Now $u' \in \text{con}(S', T')$ with $\text{start}(u) \leq \text{start}(u')$ and $\text{end}(u') = \text{end}(u)$. This means that there exists some $u'' \in U$ with $\text{start}(u) \leq \text{start}(u'')$ and $\text{end}(u'') = \text{end}(u)$, which satisfies the second constraint in the definition of *res*.

Finally, $res(\text{con}(S', T'), U)$ ensures that all instances in U have different end times. Together, this gives $res(\text{con}(S, T), U)$.

Negation case: Assume $res(\text{neg}(S', T'), U)$. Then, for any $u \in U$ we have $u \in \text{neg}(S', T')$ and thus $u \in S'$. By the subset requirement in the definition of *res*, $u \in S$. If there exists a $t \in T$ with $\text{start}(u) \leq \text{start}(t)$ and $\text{end}(t) \leq \text{end}(u)$, then there must exist some $t' \in T'$ such that $\text{start}(t) \leq \text{start}(t')$ and $\text{end}(t') = \text{end}(t)$ which contradicts the fact that $u \in \text{neg}(S', T')$. Since no such t can exist, we have $u \in \text{neg}(S, T)$ and thus $U \subseteq \text{neg}(S, T)$.

Next, take an arbitrary $u \in \text{neg}(S, T)$. Since $u \in S$ there exists an $u' \in S'$ with $\text{start}(u) \leq \text{start}(u')$, $\text{end}(u') = \text{end}(u)$. If there exists a $t \in T'$ with $\text{start}(u') \leq \text{start}(t)$ and $\text{end}(t) \leq \text{end}(u')$, then the fact that $t \in T$ contradicts $u \in \text{neg}(S, T)$. Since no such t can exist, we have that $u' \in \text{neg}(S', T')$. This means that there exists some $u'' \in U$ with $\text{start}(u') \leq \text{start}(u'')$ and $\text{end}(u'') = \text{end}(u')$, and thus $\text{start}(u) \leq \text{start}(u'')$ and $\text{end}(u'') = \text{end}(u)$, which satisfies the second constraint in the definition of *res*.

Finally, $res(\text{neg}(S', T'), U)$ ensures that all instances in U have different end times. Together, this gives $res(\text{neg}(S, T), U)$.

Sequence case: Assume $res(\text{seq}(S', T'), U)$. Then, for any $u \in U$ we have $u \in \text{seq}(S', T')$ and thus $u = s \oplus t$ with $s \in S'$, $t \in T'$ and $\text{end}(s) < \text{start}(t)$. By the subset requirement in the definition of *res*, $s \in S$ and $t \in T$, so $u \in \text{seq}(S, T)$ and thus $U \subseteq \text{seq}(S, T)$.

Next, take an arbitrary $u \in \text{seq}(S, T)$. Then $u = s \oplus t$ such that $s \in S$, $t \in T$ and $\text{end}(s) < \text{start}(t)$. By the definition of *res* there exists $s' \in S'$ and $t \in T'$ with $\text{start}(s) \leq \text{start}(s')$, $\text{end}(s') = \text{end}(s)$, $\text{start}(t) \leq \text{start}(t')$ and $\text{end}(t') = \text{end}(t)$. Let $u' = s' \oplus t'$. Now, since $\text{end}(s') = \text{end}(s) < \text{start}(t) \leq \text{start}(t')$, we have $u' \in \text{seq}(S', T')$ and $\text{start}(u) \leq \text{start}(u')$ and $\text{end}(u') =$

$\mathrm{end}(u)$. This means that there exists some $u'' \in U$ with $\mathrm{start}(u) \leq \mathrm{start}(u'')$ and $\mathrm{end}(u'') = \mathrm{end}(u)$, which satisfies the second constraint in the definition of *res*.

Finally, $\mathit{res}(\mathrm{seq}(S', T'), U)$ ensures that all instances in U have different end times. Together, this gives $\mathit{res}(\mathrm{seq}(S, T), U)$.

Temporal restriction case: Assume $\mathit{res}(\mathrm{tim}(S', \tau), U)$. For any $u \in U$ we have $u \in \mathrm{tim}(S', \tau)$ and thus $u \in S'$ and $\mathrm{end}(u) - \mathrm{start}(u) \leq \tau$. From the subset requirement in the definition of *res*, we know that $u \in S$, which means that $u \in \mathrm{tim}(S, \tau)$ and thus $U \subseteq \mathrm{tim}(S, \tau)$.

Next, take an arbitrary $u \in \mathrm{tim}(S, \tau)$. Then $u \in S$ and there exists an $u' \in S'$ with $\mathrm{start}(u) \leq \mathrm{start}(u')$, $\mathrm{end}(u') = \mathrm{end}(u)$. Since $\mathrm{end}(u) - \mathrm{start}(u) \leq \tau$, we have $\mathrm{end}(u') - \mathrm{start}(u') \leq \tau$ and thus $u' \in \mathrm{tim}(S', \tau)$. According to the def of *res*, this means that there exists some $u'' \in U$ with $\mathrm{start}(u') \leq \mathrm{start}(u'')$, $\mathrm{end}(u'') = \mathrm{end}(u')$. Since this means that $\mathrm{start}(u) \leq \mathrm{start}(u'')$, $\mathrm{end}(u'') = \mathrm{end}(u)$ the second constraint in the definition of *res* is satisfied.

Finally, $\mathit{res}(\mathrm{tim}(S', \tau), U)$ ensures that all instances in U have different end times. Together, this gives $\mathit{res}(\mathrm{tim}(S, \tau), U)$. □

To simplify the proofs for negation, we introduce the following predicate.

Definition A.1. *For an event stream S, and time instants $\tau, \tau' \in \mathcal{T}$, define $\mathit{empty}(S, \tau, \tau')$ to hold if $\neg\exists s(s \in S \wedge \tau \leq \mathrm{start}(s) \wedge \mathrm{end}(s) \leq \tau')$.*

Proposition A.1.

i. $a \in [\![A - B]\!] \;\Leftrightarrow\; (a \in [\![A]\!] \wedge \mathit{empty}([\![B]\!], \mathrm{start}(a), \mathrm{end}(a)))$.

ii. $\mathit{empty}(S \cup S', \tau, \tau') \;\Leftrightarrow\; (\mathit{empty}(S, \tau, \tau') \wedge \mathit{empty}(S', \tau, \tau'))$

iii. $(\tau_1 \leq \tau_1' \leq \tau_2' \leq \tau_2 \wedge \mathit{empty}(S, \tau_1, \tau_2)) \;\Rightarrow\; \mathit{empty}(S, \tau_1', \tau_2')$

Proof. The properties follow straightforwardly from the definition and the operator semantics. □

In the proofs below, $\equiv^{23}$ denotes that the equivalence follows from law number 23, etc. Similarly, $=^{i}$ or $\Leftrightarrow^{ii}$ denotes that the equivalence is based on the corresponding property in Proposition A.1, and $=^{\oplus}$ is based on the properties of $\oplus$ from Definition 4.4.

Theorem 4.6 For event expressions A, B and C, the following laws hold:

1.	$A \vee A \equiv A$	6.	$A;(B;C) \equiv (A;B);C$
2.	$A \vee B \equiv B \vee A$	7.	$(A \vee B)+C \equiv (A+C) \vee (B+C)$
3.	$A+B \equiv B+A$	*8.	$A+(B \vee C) \equiv (A+B) \vee (A+C)$
4.	$A \vee (B \vee C) \equiv (A \vee B) \vee C$	9.	$(A \vee B);C \equiv (A;C) \vee (B;C)$
5.	$A+(B+C) \equiv (A+B)+C$	10.	$A;(B \vee C) \equiv (A;B) \vee (A;C)$

Proof.

1. $[\![A \vee A]\!] = \mathrm{dis}([\![A]\!], [\![A]\!]) = [\![A]\!] \cup [\![A]\!] = [\![A]\!]$

2. $[\![A \vee B]\!] = \mathrm{dis}([\![A]\!], [\![B]\!]) = \mathrm{dis}([\![B]\!], [\![A]\!]) = [\![B \vee A]\!]$

3. $[\![A+B]\!] = \mathrm{con}([\![A]\!], [\![B]\!]) =^{\oplus} \mathrm{con}([\![B]\!], [\![A]\!]) = [\![B+A]\!]$

4. $[\![A \vee (B \vee C)]\!] = [\![A]\!] \cup [\![B]\!] \cup [\![C]\!] = [\![(A \vee B) \vee C]\!]$

5. $[\![A+(B+C)]\!] = \mathrm{con}([\![A]\!], \mathrm{con}([\![B]\!], [\![C]\!])) =$
   $\{a \oplus (b \oplus c) \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge c \in [\![C]\!]\} =^{\oplus}$
   $\{(a \oplus b) \oplus c \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge c \in [\![C]\!]\} = [\![(A+B)+C]\!]$

6. $[\![A;(B;C)]\!] = \{a \oplus e \mid a \in [\![A]\!] \wedge \mathrm{end}(a) < \mathrm{start}(e) \wedge$
   $e \in \{b \oplus c \mid b \in [\![B]\!] \wedge c \in [\![C]\!] \wedge \mathrm{end}(b) < \mathrm{start}(c)\}\} =$
   $\{a \oplus (b \oplus c) \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge c \in [\![C]\!] \wedge \mathrm{end}(a) < \mathrm{start}(b) \wedge$
   $\mathrm{end}(b) < \mathrm{start}(c)\} =^{\oplus}$
   $\{(a \oplus b) \oplus c) \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge c \in [\![C]\!] \wedge \mathrm{end}(a) < \mathrm{start}(b) \wedge$
   $\mathrm{end}(b) < \mathrm{start}(c)\} =$
   $\{e \oplus c \mid e \in \{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \mathrm{end}(a) < \mathrm{start}(b)\} \wedge$
   $c \in [\![C]\!] \wedge \mathrm{end}(e) < \mathrm{start}(c)\} = [\![(A;B);C]\!]$

7. $[\![(A \vee B)+C]\!] = \mathrm{con}(\mathrm{dis}([\![A]\!], [\![B]\!]), [\![C]\!]) = \mathrm{con}(([\![A]\!] \cup [\![B]\!]), [\![C]\!]) =$
   $\{e \oplus c \mid e \in [\![A]\!] \cup [\![B]\!] \wedge c \in [\![C]\!]\} =$
   $\{a \oplus c \mid a \in [\![A]\!] \wedge c \in [\![C]\!]\} \cup \{b \oplus c \mid b \in [\![A]\!] \wedge c \in [\![C]\!]\} =$
   $\mathrm{con}([\![A]\!], [\![C]\!]) \cup \mathrm{con}([\![B]\!], [\![C]\!]) = [\![(A+C) \vee (B+C)]\!]$

8. $A+(B \vee C) \equiv^{3} (B \vee C)+A \equiv^{7} (B+A) \vee (C+A) \equiv^{3} (A+B) \vee (A+C)$

9. $[\![(A \vee B);C]\!] = \{e \cup c \mid e \in [\![A]\!] \cup [\![B]\!] \wedge c \in [\![C]\!] \wedge \mathrm{end}(e) < \mathrm{start}(c)\} =$
$\{a \cup c \mid a \in [\![A]\!] \wedge c \in [\![C]\!] \wedge \mathrm{end}(a) < \mathrm{start}(c)\} \cup$
$\{b \cup c \mid b \in [\![B]\!] \wedge c \in [\![C]\!] \wedge \mathrm{end}(b) < \mathrm{start}(c)\} = [\![(A;C) \vee (B;C)]\!]$

10. $[\![A;(B \vee C)]\!] = \{a \oplus e \mid a \in [\![A]\!] \wedge e \in [\![B]\!] \cup [\![C]\!] \wedge \mathrm{end}(a) < \mathrm{start}(e)\} =$
$\{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \mathrm{end}(a) < \mathrm{start}(b)\} \cup$
$\{a \oplus c \mid a \in [\![A]\!] \wedge c \in [\![C]\!] \wedge \mathrm{end}(a) < \mathrm{start}(c)\} = [\![(A;B) \vee (A;C)]\!]$

□

Theorem 4.7 For event expressions A, B and C, the following laws hold:

$$11.\quad (A \vee B) - C \equiv (A - C) \vee (B - C)$$
$$12.\quad (A + B) - C \equiv ((A - C) + B) - C$$
$$*13.\quad (A + B) - C \equiv (A + (B - C)) - C$$
$$14.\quad (A - B) - C \equiv A - (B \vee C)$$
$$*15.\quad (A - B) - B \equiv A - B$$
$$*16.\quad (A - B) - C \equiv (A - C) - B$$
$$17.\quad (A;B) - C \equiv ((A - C);B) - C$$
$$18.\quad (A;B) - C \equiv (A;(B - C)) - C$$

Proof.

11. $[\![(A \vee B) - C]\!] =^i \{e \mid e \in [\![A]\!] \cup [\![B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e))\} =$
$\{a \mid a \in [\![A]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(a))\} \cup$
$\{b \mid b \in [\![B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(b), \mathrm{end}(b))\} =^i$
$[\![(A - C)]\!] \cup [\![(B - C)]\!] = [\![(A - C) \vee (B - C)]\!]$

12. $e \in [\![((A - C) + B) - C]\!] \Leftrightarrow^i e \in [\![(A - C) + B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow$
$e = a \oplus b \wedge a \in [\![A - C]\!] \wedge b \in [\![B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow^i$
$e = a \oplus b \wedge a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \wedge$
$\mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(a)) \Leftrightarrow^{iii}$
$e = a \oplus b \wedge a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow$
$e \in [\![A + B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow^i e \in [\![(A + B) - C]\!]$

13. $(A + B) - C \equiv^3 (B + A) - C \equiv^{12} ((B - C) + A) - C \equiv^3 (A + (B - C)) - C$

14. $a \in [\![(A - B) - C]\!] \Leftrightarrow^i a \in [\![A - B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(a)) \Leftrightarrow^i$
$a \in [\![A]\!] \wedge \mathit{empty}([\![B]\!], \mathrm{start}(a), \mathrm{end}(a)) \wedge \mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(a)) \Leftrightarrow^{ii}$
$a \in [\![A]\!] \wedge \mathit{empty}([\![B]\!] \cup [\![C]\!], \mathrm{start}(a), \mathrm{end}(a)) \Leftrightarrow^i a \in [\![A - (B \vee C)]\!]$

15. $(A-B)-B \equiv^{14} A-(B\vee B) \equiv^{1} A-B$

16. $(A-B)-C \equiv^{14} A-(B\vee C) \equiv^{2} A-(C\vee B) \equiv^{14} (A-C)-B$

17. $e\in[\![((A\text{–}C);B)\text{–}C]\!] \Leftrightarrow^{i} e\in[\![(A\text{–}C);B]\!]\wedge \mathit{empty}([\![C]\!],\mathrm{start}(e),\mathrm{end}(e)) \Leftrightarrow$
$e=a\oplus b\wedge \mathrm{end}(a)<\mathrm{start}(b)\wedge a\in[\![A-C]\!]\wedge b\in[\![B]\!]\wedge$
$\mathit{empty}([\![C]\!],\mathrm{start}(a),\mathrm{end}(b)) \Leftrightarrow^{i}$
$e=a\oplus b\wedge \mathrm{end}(a)<\mathrm{start}(b)\wedge a\in[\![A]\!]\wedge b\in[\![B]\!]\wedge$
$\mathit{empty}([\![C]\!],\mathrm{start}(a),\mathrm{end}(b))\wedge \mathit{empty}([\![C]\!],\mathrm{start}(a),\mathrm{end}(a)) \Leftrightarrow^{iii}$
$e=a\oplus b\wedge \mathrm{end}(a)<\mathrm{start}(b)\wedge a\in[\![A]\!]\wedge b\in[\![B]\!]\wedge$
$\mathit{empty}([\![C]\!],\mathrm{start}(a),\mathrm{end}(b)) \Leftrightarrow$
$e\in[\![A;B]\!]\wedge \mathit{empty}([\![C]\!],\mathrm{start}(e),\mathrm{end}(e)) \Leftrightarrow^{i} e\in[\![(A;B)-C]\!]$

18. $e\in[\![(A;(B\text{–}C))\text{–}C]\!] \Leftrightarrow^{i} e\in[\![A;(B\text{–}C)]\!]\wedge \mathit{empty}([\![C]\!],\mathrm{start}(e),\mathrm{end}(e)) \Leftrightarrow$
$e=a\oplus b\wedge \mathrm{end}(a)<\mathrm{start}(b)\wedge a\in[\![A]\!]\wedge b\in[\![B-C]\!]\wedge$
$\mathit{empty}([\![C]\!],\mathrm{start}(a),\mathrm{end}(b)) \Leftrightarrow^{i}$
$e=a\oplus b\wedge \mathrm{end}(a)<\mathrm{start}(b)\wedge a\in[\![A]\!]\wedge b\in[\![B]\!]\wedge$
$\mathit{empty}([\![C]\!],\mathrm{start}(a),\mathrm{end}(b))\wedge \mathit{empty}([\![C]\!],\mathrm{start}(b),\mathrm{end}(b)) \Leftrightarrow^{iii}$
$e=a\oplus b\wedge \mathrm{end}(a)<\mathrm{start}(b)\wedge a\in[\![A]\!]\wedge b\in[\![B]\!]\wedge$
$\mathit{empty}([\![C]\!],\mathrm{start}(a),\mathrm{end}(b)) \Leftrightarrow$
$e\in[\![A;B]\!]\wedge \mathit{empty}([\![C]\!],\mathrm{start}(e),\mathrm{end}(e)) \Leftrightarrow^{i} e\in[\![(A;B)-C]\!]$

□

Theorem 4.8 For event expressions A and B, and $\tau\in\mathcal{T}$, the following laws hold:

19.	$(A\vee B)_\tau \equiv A_\tau\vee B_\tau$	24.	$(A;B)_\tau \equiv (A_\tau;B)_\tau$
20.	$(A+B)_\tau \equiv (A_\tau+B)_\tau$	25.	$(A;B)_\tau \equiv (A;B_\tau)_\tau$
*21.	$(A+B)_\tau \equiv (A+B_\tau)_\tau$	26.	$A \equiv A_\tau$ if $A\in\mathcal{P}$
22.	$(A-B)_\tau \equiv A_\tau-B$	27.	$(A_\tau)_{\tau'} \equiv A_{\min(\tau,\tau')}$
23.	$(A-B)_\tau \equiv (A-B_\tau)_\tau$	*28.	$(A_\tau)_{\tau'} \equiv (A_{\tau'})_\tau$

Proof.

19. $[\![(A\vee B)_\tau]\!] = \{e \mid e\in A\cup B\wedge \mathrm{end}(e)-\mathrm{start}(e)\le\tau\} =$
$\{a \mid a\in A\wedge \mathrm{end}(a)-\mathrm{start}(a)\le\tau\}\cup\{b \mid b\in B\wedge \mathrm{end}(b)-\mathrm{start}(b)\le$
$\tau\} = [\![A_\tau]\!]\cup[\![B_\tau]\!] = [\![A_\tau\vee B_\tau]\!]$

20. $e \in [\![(A_\tau + B)_\tau]\!] \Leftrightarrow e \in [\![A_\tau + B]\!] \wedge \mathrm{end}(e) - \mathrm{start}(e) \leq \tau \Leftrightarrow$
$e = a \oplus b \wedge a \in [\![A_\tau]\!] \wedge b \in [\![B]\!] \wedge \mathrm{end}(e) - \mathrm{start}(e) \leq \tau \Leftrightarrow$
$e = a \oplus b \wedge a \in [\![A]\!] \wedge \mathrm{end}(a) - \mathrm{start}(a) \leq \tau \wedge b \in [\![B]\!] \wedge \mathrm{end}(e) - \mathrm{start}(e) \leq \tau$.
Since $\mathrm{end}(a) \leq \mathrm{end}(e)$ and $\mathrm{start}(e) \leq \mathrm{start}(a)$, we have $\mathrm{end}(a) - \mathrm{start}(a) \leq \mathrm{end}(e) - \mathrm{start}(e)$, so $\mathrm{end}(e) - \mathrm{start}(e) \leq \tau \Rightarrow \mathrm{end}(a) - \mathrm{start}(a) \leq \tau$. Thus, the last formula above is equivalent to:
$e = a \oplus b \wedge a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \mathrm{end}(e) - \mathrm{start}(e) \leq \tau \Leftrightarrow$
$e \in [\![A_\tau + B]\!] \wedge \mathrm{end}(e) - \mathrm{start}(e) \leq \tau \Leftrightarrow e \in [\![(A + B)_\tau]\!]$.

21. $(A+B)_\tau \equiv^3 (B+A)_\tau \equiv^{20} (B_\tau + A)_\tau \equiv^3 (A + B_\tau)_\tau$

22. $[\![(A-B)_\tau]\!] = \{a \mid a \in [\![A - B]\!] \wedge \mathrm{end}(a) - \mathrm{start}(a) \leq \tau\} =$
$\{a \mid a \in [\![A]\!] \wedge \mathit{empty}([\![B]\!], \mathrm{start}(a), \mathrm{end}(a)) \wedge \mathrm{end}(a) - \mathrm{start}(a) \leq \tau\} =$
$\{a \mid a \in [\![A_\tau]\!] \wedge \mathit{empty}([\![B]\!], \mathrm{start}(a), \mathrm{end}(a))\} = [\![A_\tau - B]\!]$

23. $[\![(A - B_\tau)_\tau]\!] = \{a \mid a \in [\![A]\!] \wedge \mathrm{end}(a) - \mathrm{start}(a) \leq \tau \wedge$
$\neg \exists b (b \in [\![B_\tau]\!] \wedge \mathrm{start}(a) \leq \mathrm{start}(b) \wedge \mathrm{end}(b) \leq \mathrm{end}(a))\} =$
$\{a \mid a \in [\![A]\!] \wedge \mathrm{end}(a) - \mathrm{start}(a) \leq \tau \wedge \neg \exists b (b \in [\![B]\!] \wedge$
$\mathrm{start}(a) \leq \mathrm{start}(b) \wedge \mathrm{end}(b) \leq \mathrm{end}(a) \wedge \mathrm{end}(b) - \mathrm{start}(b) \leq \tau)\}$
Since $\mathrm{end}(a) - \mathrm{start}(a) \leq \tau$, $\mathrm{start}(a) \leq \mathrm{start}(b)$ and $\mathrm{end}(b) \leq \mathrm{end}(a)$ implies $\mathrm{end}(b) - \mathrm{start}(b) \leq \tau$, that constraint can be removed without affecting the set. Thus, the set above is equivalent to:
$\{a \mid a \in [\![A]\!] \wedge \mathrm{end}(a) - \mathrm{start}(a) \leq \tau \wedge$
$\neg \exists b (b \in [\![B]\!] \wedge \mathrm{start}(a) \leq \mathrm{start}(b) \wedge \mathrm{end}(b) \leq \mathrm{end}(a))\} = [\![(A - B)_\tau]\!]$.

24. $[\![(A;B_\tau)_\tau]\!] =$
$\{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B_\tau]\!] \wedge \mathrm{end}(a) < \mathrm{start}(b) \wedge \mathrm{end}(b) - \mathrm{start}(a) \leq \tau\} =$
$\{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \mathrm{end}(b) - \mathrm{start}(b) \leq \tau \wedge \mathrm{end}(a) < \mathrm{start}(b) \wedge$
$\mathrm{end}(b) - \mathrm{start}(a) \leq \tau\}$
Since $\mathrm{end}(a) < \mathrm{start}(b)$ and $\mathrm{end}(b) - \mathrm{start}(a) \leq \tau$ implies $\mathrm{end}(b) - \mathrm{start}(b) \leq \tau$, this constraint can be dropped without changing the set. Thus, the set above is equivalent to $\{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \mathrm{end}(a) < \mathrm{start}(b) \wedge \mathrm{end}(b) - \mathrm{start}(a) \leq \tau\} = [\![(A;B)_\tau]\!]$

25. $[\![(A_\tau;B)_\tau]\!] =$
$\{a \oplus b \mid a \in [\![A_\tau]\!] \wedge b \in [\![B]\!] \wedge \mathrm{end}(a) < \mathrm{start}(b) \wedge \mathrm{end}(b) - \mathrm{start}(a) \leq \tau\} =$
$\{a \oplus b \mid a \in [\![A]\!] \wedge \mathrm{end}(a) - \mathrm{start}(a) \leq \tau \wedge b \in [\![B]\!] \wedge \mathrm{end}(a) < \mathrm{start}(b) \wedge \mathrm{end}(b) - \mathrm{start}(a) \leq \tau\}$
Since $\mathrm{end}(a) < \mathrm{start}(b)$ and $\mathrm{end}(b) - \mathrm{start}(a) \leq \tau$ implies $\mathrm{end}(a) -$

start$(a) \leq \tau$, this constraint can be dropped without changing the set. Thus, the set above is equivalent to $\{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \text{end}(a) < \text{start}(b) \wedge \text{end}(b) - \text{start}(a) \leq \tau\} = [\![(A;B)_\tau]\!]$

26. $A \in \mathcal{P}$ implies that $\text{end}(a) - \text{start}(a) = 0$ for any $a \in [\![A]\!]$, which means that $[\![A]\!] = [\![A_\tau]\!]$.

27. $[\![(A_\tau)_{\tau'}]\!] = \{a \mid a \in [\![A]\!] \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge \text{end}(a) - \text{start}(a) \leq \tau'\} = \{a \mid a \in [\![A]\!] \wedge \text{end}(a) - \text{start}(a) \leq \min(\tau, \tau')\} = [\![A_{\min(\tau,\tau')}]\!]$

28. $(A_\tau)_{\tau'} \equiv^{23} A_{\min(\tau,\tau')} \equiv A_{\min(\tau,\tau')} \equiv^{23} (A_{\tau'})_\tau$

□

Lemma 5.2 Assume that *state*(i, τ) held at the start of the current tick and that *pcorr*(n, τ) and *acorr*(n, τ) hold for all $1 \leq n < i$. Then *state*$(i, \tau + 1)$, *pcorr*(i, τ) and *acorr*(i, τ) hold after executing the loop body once.

Proof. The proof is organised in four parts. First, we consider *state*, then the two criteria that are required for *pcorr* to hold (see Definition 5.2), and finally *acorr* is addressed.

For *state*, we see that *state*$(i, \tau + 1)$ holds trivially if E^i is primitive, a disjunction or a temporal restriction (Definition 5.3), and thus we consider the remaining operators:

Case $E^i = E^j + E^k$: In the case $a_j = \varepsilon$ the l_i variable remains unchanged, which is consistent with *state*$(i, \tau + 1)$. If $a_j \neq \varepsilon$ then $\text{end}(a_j) = \tau$ according to the assumption *acorr*(j, τ). Then, the first conditional in the conjunction part ensures that l_i contains an instance consistent with *state*$(i, \tau+1)$. Similarly, the second conditional ensures the correctness of r_i.

Case $E^i = E^j - E^k$: The first conditional in the negation part ensures that t_i contains the value specified by *state*$(i, \tau+1)$.

Case $E^i = E^j;E^k$: The l_i variable is updated by the last conditional in the sequence part, and the proof is identical to that in the conjunction case above. For the second criterion in the definition of *state*, let t be an arbitrary element in S_k such that $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < t\}$ is non-empty. We consider two cases: If t was in S_k at the start of the current tick, then *state*(i, τ) ensures that Q_i contained an element from $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < t\}$ with maximum start time at the start of this tick. If t was not in S_k at the start of the current tick, then *pcorr*(k, τ)

implies that $t = \tau$, and then $state(i, \tau)$ ensures that l_i contained an element from $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < t\}$ with maximum start time at the start of this tick. Thus, in both cases, $Q_i \cup l_i$ contained such an element at the start of this tick. We can see that the inner foreach construct in the sequence part assigns an element from this set to e', and thus it is added to Q' and finally to Q_i.

For *pcorr*, let S denote the content of S_i at the start of the current time tick. We focus first on the first criterion in the definition of *pcorr*, which requires that we have $a_i = \varepsilon$, $\text{start}(a_i) = \tau$ or $\text{start}(a_i) \in S$. For $E^i \in \mathcal{P}$, we know that a_i is a primitive event instance, and thus $\text{start}(a_i) = \text{end}(a_i) = \tau$. For the operators, we note that the first criterion of $pcorr(i, \tau)$ holds trivially when $a_i = \varepsilon$ or $\text{start}(a_i) = \tau$, so we consider only the case when $a_i \neq \varepsilon$ and $\text{start}(a_i) \neq \tau$.

Case $E^i = E^j \vee E^k$: If $a_i = a_j$, we know according to $pcorr(j, \tau)$ that $\text{start}(a_i)$ was in S_j at the start of this tick. Since $S_j \subseteq S_i$ must hold at the start of each tick (at initialisation, and after each subsequent assignment of S_i), this implies $\text{start}(a_i) \in S$. If $a_i = a_k$, the same result is implied by $pcorr(k, \tau)$ and $S_k \subseteq S_i$.

Case $E^i = E^j + E^k$: From the two assignments of a_i in the conjunction part where $a_i \neq \varepsilon$, we can see that the start time of a_i must be equal to the start time of a_j, r_i, l_i or a_k. For a_j and a_k we can reuse the disjunction proof above. If $\text{start}(a_i) = \text{start}(l_i)$, we have to consider two subcases: If l_i was updated in this tick, we have $l_i = a_j$ and we can reuse the proof above. If l_i remained unchanged, then $l_i \neq \varepsilon$ ensures that the current tick is not the first, and the assignment of S_i in the previous step implies that $\text{start}(l_i) \in S$. The proof for the final case $\text{start}(a_i) = \text{start}(r_i)$ is analogous.

Case $E^i = E^j - E^k$: Analogous to the $a_i = a_j$ case in the disjunction proof.

Case $E^i = E^j ; E^k$: Since $a_i \neq \varepsilon$, we have $a_i = a_k \oplus e'$ where $\text{start}(a_i) = \text{start}(e')$ and e' was in Q_i or l_i at the start of this tick. This implies that the current tick is not the first, and the assignment of S_i in the previous step ensures that $\text{start}(e') \in S$.

Case $E^i = E^j_{\tau'}$: Analogous to the $a_i = a_j$ case in the disjunction proof.

Next, we consider the second criterion in the definition of *pcorr*, namely that $\forall t \ (t \in S_i \Rightarrow (t = \tau \vee t \in S))$. As previously, S denotes the content of S_i at the start of the current time tick. The property trivially holds for $E^i \in \mathcal{P}$, since this implies $S = \emptyset$. For the operators, consider

an arbitrary $t \in S_i$ such that $t \neq \tau$.

Case $E^i = E^j \vee E^k$: Since $S_i = S_j \cup S_k$, we must have $t \in S_j$ or $t \in S_k$. If $t \in S_j$, then $\mathit{pcorr}(j, \tau)$ implies that t was in S_j at the start of this tick. Since $S_j \subseteq S_i$ holds at the start of each tick, this implies $t \in S$. If $t \in S_k$, the same result follows from $\mathit{pcorr}(k, \tau)$ and $S_k \subseteq S_i$.

Case $E^i = E^j + E^k$: The assignment of S_i in the conjunction part implies that $t \in S_j \cup S_k \cup \{\mathrm{start}(l_i), \mathrm{start}(r_i)\}$. If $t \in S_j \cup S_k$, we can reuse the disjunction proof above. If $t = \mathrm{start}(l_i)$ we consider two subcases: If l_i remained unchanged in this tick, then the assignment of S_i in the previous tick ensures than $t \in S$. If l_i was updated, we have $l_i = a_j$, and then $\mathit{pcorr}(j, \tau)$ ensures that t was in S_j at the start of this tick. As shown above, this implies $t \in S$. The proof for the final case $t = \mathrm{start}(r_i)$ is analogous.

Case $E^i = E^j - E^k$: Analogous to the $t \in S_j$ case in the disjunction proof.

Case $E^i = E^j ; E^k$: The assignment of S_i in the sequence part implies that $t \in S_j$, $t = \mathrm{start}(l_i)$ or $t \in \{\mathrm{start}(e) \mid e \in Q_i\}$. For the two first cases we can reuse the proof for conjunction. If $t = \mathrm{start}(e)$ where $e \in Q_i$, then e was added to Q' in the nested foreach constructs, which means that e was in $Q_i \cup l_i$ at the start of this tick (so this is not the first tick). Then, the assignment of S_i in the previous tick ensures than $t \in S$.

Case $E^i = E^j_{\tau'}$: Analogous to the $t \in S_j$ case in the disjunction proof.

Finally, for *acorr*, we consider the following six cases:

Case $E^i \in \mathcal{P}$: If $a_i = \varepsilon$, then there is no $e \in [\![E^i]\!]$ with $\mathrm{end}(e) = \tau$, and thus $\mathit{valid}(a_i, [\![E^i]\!], \tau)$ holds. If $a_i \neq \varepsilon$, we have $a_i \in [\![E^i]\!]$ and $\mathrm{end}(a_i) = \tau$, and since the elements of $[\![E^i]\!]$ have distinct end times according to Definition 4.5, $\mathit{valid}(a_i, [\![E^i]\!], \tau)$ holds.

Case $E^i = E^j \vee E^k$: The detection algorithm ensures that $\mathrm{start}(a_j) \leq \mathrm{start}(a_i)$ and $\mathrm{start}(a_k) \leq \mathrm{start}(a_i)$. If $a_i = \varepsilon$, we have $\mathrm{start}(a_i) = -1$ which implies that $a_k = a_j = \varepsilon$ so there is no $e \in \mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k))$ with $\mathrm{end}(e) = \tau$. If $a_i \neq \varepsilon$, we clearly have $a_i \in \mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k))$ and there can be no element in this set with end time τ and start time later than $\mathrm{start}(a_i)$.

Case $E^i = E^j + E^k$: After executing the first two conditionals in the conjunction part $\mathrm{start}(a_j) \leq \mathrm{start}(l_i)$ and $\mathrm{start}(a_k) \leq \mathrm{start}(r_i)$ hold. If $a_i = \varepsilon$, then the guard of the third conditional was satisfied, and there can be no instance in $\mathrm{con}(\mathcal{A}(j), \mathcal{A}(k))$ with end time τ, which concludes the proof. If $a_i \neq \varepsilon$, then the guard of the third conditional failed,

and the inner conditional ensures that $a_i \in \mathrm{con}(\mathcal{A}(j), \mathcal{A}(k))$. For an arbitrary $e \in \mathrm{con}(\mathcal{A}(j), \mathcal{A}(k))$ with $\mathrm{end}(e) = \tau$, we must have $e = e' \oplus a_k$ or $e = a_j \oplus e'$ where $e' \in \mathcal{A}(j) \cup \mathcal{A}(k)$ and $\mathrm{end}(e') \leq \tau$. However, the inner conditional ensures that $\mathrm{start}(a_j) \leq \mathrm{start}(a_i)$ and $\mathrm{start}(a_k) \leq \mathrm{start}(a_i)$ which implies $\mathrm{start}(e) \leq \mathrm{start}(a_i)$, and thus there is no $e \in \mathrm{con}(\mathcal{A}(j), \mathcal{A}(k))$ with $\mathrm{end}(e) = \tau$ and $\mathrm{start}(a_i) < \mathrm{start}(e)$.

Case $E^i = E^j - E^k$: Reusing the proof for *state* above, we know that $\mathit{state}(i, \tau{+}1)$ holds after the first conditional in the negation part. If $a_i = \varepsilon$, then the guard of the second conditional failed, implying that either $a_j = \varepsilon$ or there exists an e in $\mathcal{A}(k)$ with $\mathrm{start}(a_j) \leq \mathrm{start}(e)$ and $\mathrm{end}(e) \leq \mathrm{end}(a_j)$. In either case, there is no element in $\mathrm{neg}(\mathcal{A}(j), \mathcal{A}(k))$ with end time τ. If $a_i \neq \varepsilon$, then $a_i = a_j$ so we have $a_i \in \mathcal{A}(j)$. Furthermore, the guard of the second conditional holds and then according to $\mathit{state}(i, \tau{+}1)$ there is no e in $\mathcal{A}(k)$ with $\mathrm{start}(a_i) \leq \mathrm{start}(e)$ and $\mathrm{end}(e) \leq \mathrm{end}(a_i)$, and thus $a_i \in \mathrm{neg}(\mathcal{A}(j), \mathcal{A}(k))$. Since a_j is the only instance in $\mathcal{A}(j)$ with end time τ, we have $\mathit{valid}(a_i, \mathrm{neg}(\mathcal{A}(j), \mathcal{A}(k), \tau)$.

Case $E^i = E^j; E^k$: If $a_k = \varepsilon$ then $e' = \varepsilon$ after the first foreach construct, and thus $a_i = \varepsilon$. It also means that there can be no $e \in \mathrm{seq}(\mathcal{A}(j), \mathcal{A}(k))$ with $\mathrm{end}(e) = \tau$, which concludes the proof. If $a_k \neq \varepsilon$ then $\mathit{pcorr}(k, \tau)$ implies that either $\mathrm{start}(a_k) = \tau$ or $\mathrm{start}(a_k)$ was in S_k at the start of this tick. According to $\mathit{state}(i, \tau)$, this implies that (at the start of this tick) $Q_i \cup \{l_i\}$ contained an element in $\{e \mid e \in \mathcal{A}(j) \wedge \mathrm{end}(e) < \mathrm{start}(a_k)\}$ with maximum start time if that set is non-empty. We consider two subcases: If $e' = \varepsilon$ after the first foreach contruct, the set was empty, meaning that there can be no element in $e \in \mathrm{seq}(\mathcal{A}(j), \mathcal{A}(k))$ with $\mathrm{end}(e) = \tau$. If $e' \neq \varepsilon$ after the first foreach contruct, then $a_i = a_k \oplus e'$ ensures that $a_i \in \mathrm{seq}(\mathcal{A}(j), \mathcal{A}(k))$. Furthermore, we know that there is no $e \in \mathcal{A}(j)$ with $\mathrm{end}(e) < \mathrm{start}(a_k)$ and $\mathrm{start}(e') < \mathrm{start}(e)$. Thus, there can be no $e \in \mathrm{seq}(\mathcal{A}(j), \mathcal{A}(k))$ with $\mathrm{start}(a_i) < \mathrm{start}(e)$ and $\mathrm{end}(e) = \tau$.

Case $E^i = E^j_{\tau'}$: If the conditional holds, we have $a_j \in \mathrm{tim}(\mathcal{A}(j), \tau')$. Since a_j is the only instance in $\mathcal{A}(j)$ that has end time τ, we have $\mathit{valid}(a_j, \mathrm{tim}(\mathcal{A}(j), \tau'), \tau)$. If the conditional fails, then there is no e in $\mathrm{tim}(\mathcal{A}(j), \tau')$ with $\mathrm{end}(e) = \tau$. □

Appendix B

Memory and Time Analysis Algorithm

Section 5.5 described how abstract notions of memory footprint and worst case execution time can be derived directly from a given event expression and the instance framework. Here, the whole analysis algorithm is presented.

If analyse(E) returns $\langle m, t \rangle$, this means that the detection of E requires m units of memory and takes t units of time to execute. The auxiliary function analyse_aux(E, sn) includes the boolean parameter sn to indicate whether or not the S_i variable is needed for the principal operator of E (they are only needed within the right-hand subexpression of a sequence operator). The first two elements of the four-tuple $\langle s, i, m, t \rangle$ returned from the auxiliary function capture properties related to the principal operator of E, namely the size of S_i and the instance size. The other two represent the memory usage and execution time of the whole expression E.

$\text{analyse}(E) = \langle m+1,\ t+2\rangle$
where
$\langle s, i, m, t\rangle = \text{analyse\_aux}(E,\ \text{false})$

$\text{analyse\_aux}(E,\ sn) = \langle 0, i, 1+i, 4+i\rangle$
when $E \in \mathcal{P}$
where
$i = \text{primsize}(E^i)$

$\text{analyse\_aux}(E^j \vee E^k,\ sn) = \langle s, i, m, t\rangle$
where
$\langle s_j, i_j, m_j, t_j\rangle = \text{analyse\_aux}(E^j,\ sn)$
$\langle s_k, i_k, m_k, t_k\rangle = \text{analyse\_aux}(E^k,\ sn)$
if sn then $s = s_j + s_k$ else $s = 0$
$i = \text{unionsize}(i_j,\ i_k)$
$m = m_j + m_k + 1 + s + i$
$t = t_j + t_k + 5 + s + i$

$\text{analyse\_aux}(E^j + E^k,\ sn) = \langle s, i, m, t\rangle$
where
$\langle s_j, i_j, m_j, t_j\rangle = \text{analyse\_aux}(E^j,\ sn)$
$\langle s_k, i_k, m_k, t_k\rangle = \text{analyse\_aux}(E^k,\ sn)$
if sn then $s = s_j + s_k + 2$ else $s = 0$
$i = \text{compsize}(i_j,\ i_k)$
$m = m_j + m_k + 1 + s + i + i_j + i_k$
$t = t_j + t_k + 14 + s + i + i_j + i_k$

$\text{analyse\_aux}(E^j - E^k,\ sn) = \langle s, i, m, t\rangle$
where
$\langle s_j, i_j, m_j, t_j\rangle = \text{analyse\_aux}(E^j,\ sn)$
$\langle s_k, i_k, m_k, t_k\rangle = \text{analyse\_aux}(E^k,\ sn)$
if sn then $s = s_j$ else $s = 0$
$i = i_j$
$m = m_j + m_k + 1 + s + i$
$t = t_j + t_k + 7 + s + i$

analyse_aux$(E^j;E^k,\ sn) = \langle s, i, m, t\rangle$
where
$\langle s_j, i_j, m_j, t_j\rangle$ = analyse_aux$(E^j,\ sn)$
$\langle s_k, i_k, m_k, t_k\rangle$ = analyse_aux$(E^k,\ \text{true})$
if sn then $s = s_j + s_k + 1$ else $s = 0$
i = compsize$(i_j,\ i_k)$
$m = m_j + m_k + 4 + s + i + (4 + 2 * s_k) * i_j$
$t = t_j + t_k + 20 + 19 * s_k + s + i + (2 + 5 * s_k) * i_j$

analyse_aux$((E^j)_\tau,\ sn) = \langle s, i, m, t\rangle$
where
$\langle s_j, i_j, m_j, t_j\rangle$ = analyse_aux$(E^j,\ sn)$
if sn then $s = s_j$ else $s = 0$
$i = i_j$
$m = m_j + 1 + s + i$
$t = t_j + 6 + s + i$

Appendix C

Schedulability Analysis Examples

This appendix presents the schedulability examples from Chapter 6 (i.e., Examples 6.3 and 6.5) in more detail. We consider a system with three tasks, two of which are periodic, and one which is triggered by the event expression $(A;B)+C$.

Task ID	C_i	T_i	D_i	E_i
T1	10	50	30	–
P2	20	–	100	$(A;B)+C$
T3	30	200	200	–

The schedulability analysis, also requires information about the minimum interarrival time of the primitive events, as well as the worst case execution time associated with detecting the pattern. We assume that $\text{mint}(A) = 60$, $\text{mint}(B) = 70$, $\text{mint}(C) = 200$ and $\text{wcet}((A;B)+C) = 5$.

Following Definition 6.4, an auxiliary task set is generated from the original task set and the additional information.

From task	t_i	C_i	T_i	D_i
T1	t_1	10	50	30
P2	t_2	5	60	100
	t_3	25	70	100
	t_4	25	200	100
T3	t_5	30	200	200

The original task set can not demand more computational resources over time than this auxiliary task set, and since the latter consists only of periodic and sporadic tasks, it can be analysed with standard techniques. If the auxiliary task set is schedulable, then so is the original task set.

Response time analysis

For the fixed priority schedulability analysis, we modified the response time analysis from Tindell et al. [147] to take into account that tasks do not have unique priorities.

$$w_i(q) = \sum_{\forall j \in ep(i)} \left(\left\lfloor \frac{qT_i}{T_j} \right\rfloor + 1 \right) C_j + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i(q)}{T_j} \right\rceil C_j$$

$$r_i = \max_{q=0,1,\ldots,\left\lfloor \frac{L(P_i)}{T_i} \right\rfloor} (w_i(q) - qT_i)$$

To determine the smallest value of $w_i(q)$ that satisfies the equation above, a sequence of values is generated as follows:

$$w_i^1(q) = \sum_{\forall j \in ep(i)} \left(\left\lfloor \frac{qT_i}{T_j} \right\rfloor + 1 \right) C_j$$

$$w_i^{n+1}(q) = w_i^1(q) + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j$$

The iteration ends when a fixed point is found, or when a value greater than $qT_i + D_i$ is encountered, which indicates that the response time exceeds the deadline.

For the example task set, we assume that T1 has highest priority, followed by P2 and finally T3. These priorities propagate to the auxiliary task set:

From task	i	P_i	C_i	T_i	D_i	r_i
T1	1	High	10	50	30	10
P2	2	Mid	5	60	100	75
	3	Mid	25	70	100	75
	4	Mid	25	200	100	75
T3	5	Low	30	200	200	190

Since $r_i \leq D_i$ for all tasks, the original task set is schedulable. The remaining section describes how the response times are derived.

Task t_1

For t_1, we have $L(P_1) = 10$ and $\lfloor 10/50 \rfloor = 0$, meaning that only one instance of t_1 can arrive during a high priority busy period. Since no other task instance can interfer with this instance, the response time of task t_1 is $r_1 = w_1(0) = C_j = 10$.

Task t_2

For t_2 we have $L(P_2) = 115$, and $\lfloor 115/60 \rfloor = 1$, which means that at most two instances can arrive during a middle priority busy period. The latest start time for instance q is given by

$$\begin{aligned} w_2^1(q) &= (q+1)5 + \left(\left\lfloor \tfrac{60q}{70} \right\rfloor + 1\right) 25 + \left(\left\lfloor \tfrac{60q}{200} \right\rfloor + 1\right) 25 \\ w_2^{n+1}(q) &= w_2^1(q) + \left\lceil \tfrac{w_2^n(q)}{50} \right\rceil 10 \end{aligned}$$

Fixpoint iteration for $q = 0$:

$$\begin{aligned} w_2^1(0) &= 5 + 25 + 25 = 55 \\ w_2^2(0) &= 55 + \lceil 55/50 \rceil 10 = 75 \\ w_2^3(0) &= 55 + \lceil 75/50 \rceil 10 = 75 \end{aligned}$$

Fixpoint iteration for $q = 1$:

$$\begin{aligned} w_2^1(1) &= 10 + 25 + 25 = 60 \\ w_2^2(1) &= 60 + \lceil 60/50 \rceil 10 = 80 \\ w_2^3(1) &= 60 + \lceil 80/50 \rceil 10 = 80 \end{aligned}$$

The response time is $r_2 = \max(w_2(0), w_2(1) - 60) = \max(75, 20) = 75$.

Task t_3

For t_3 we have $L(P_3) = L(P_2) = 115$ and $\lfloor 115/70 \rfloor = 1$, which means that at most two instances can arrive during a middle priority busy period. The latest start time for instance q is given by

$$\begin{aligned} w_3^1(q) &= (q+1)25 + \left(\left\lfloor \tfrac{70q}{60} \right\rfloor + 1\right) 5 + \left(\left\lfloor \tfrac{70q}{200} \right\rfloor + 1\right) 25 \\ w_3^{n+1}(q) &= w_3^1(q) + \left\lceil \tfrac{w_3^n(q)}{50} \right\rceil 10 \end{aligned}$$

The latest finish time for instance $q = 0$ is the same for all tasks with the same priority. Thus, we have $w_3(0) = w_2(0) = 75$.
Fixpoint iteration for $q = 1$:

$$\begin{aligned} w_3^1(1) &= 50 + 10 + 25 = 85 \\ w_3^2(1) &= 85 + \lceil 85/50 \rceil 10 = 105 \\ w_3^3(1) &= 85 + \lceil 105/50 \rceil 10 = 115 \\ w_3^4(1) &= 85 + \lceil 115/50 \rceil 10 = 115 \end{aligned}$$

The response time is $r_3 = \max(w_3(0), w_3(1) - 70) = \max(75, 45) = 75$.

Task t_4

For t_4 we have $L(P_4) = L(P_2) = 115$ and $\lfloor 115/200 \rfloor = 0$, which means that at most one instance can arrive during a middle priority busy period. The latest finish time for instance $q = 0$ is the same for all tasks with the same priority. Thus, $w_4(0) = w_2(0) = 75$, and since there is only one instance of t_4 to investigate, we have $r_4 = w_4(0) = 75$.

Task t_5

For t_5 we have $L(P_5) = 190$ and $\lfloor 190/200 \rfloor = 0$, which means that at most one instance can arrive during a low priority busy period. The latest start time for instance q is given by

$$\begin{aligned} w_5^1(q) &= (q+1)30 \\ w_5^{n+1}(q) &= w_5^1(q) + \left\lceil \tfrac{w_5^n(q)}{50} \right\rceil 10 + \left\lceil \tfrac{w_5^n(q)}{60} \right\rceil 5 + \left\lceil \tfrac{w_5^n(q)}{70} \right\rceil 25 + \left\lceil \tfrac{w_5^n(q)}{200} \right\rceil 25 \end{aligned}$$

Fixpoint iteration for $q = 0$:

$$\begin{aligned}
w_5^1(0) &= 30 \\
w_5^2(0) &= 30 + \lceil \tfrac{30}{50} \rceil 10 + \lceil \tfrac{30}{60} \rceil 5 + \lceil \tfrac{30}{70} \rceil 25 + \lceil \tfrac{30}{200} \rceil 25 \\
&= 30 + 10 + 5 + 25 + 25 = 95 \\
w_5^3(0) &= 30 + \lceil \tfrac{95}{50} \rceil 10 + \lceil \tfrac{95}{60} \rceil 5 + \lceil \tfrac{95}{70} \rceil 25 + \lceil \tfrac{95}{200} \rceil 25 \\
&= 30 + 20 + 10 + 50 + 25 = 135 \\
w_5^4(0) &= 30 + \lceil \tfrac{135}{50} \rceil 10 + \lceil \tfrac{135}{60} \rceil 5 + \lceil \tfrac{135}{70} \rceil 25 + \lceil \tfrac{135}{200} \rceil 25 \\
&= 30 + 30 + 15 + 50 + 25 = 150 \\
w_5^5(0) &= 30 + \lceil \tfrac{150}{50} \rceil 10 + \lceil \tfrac{150}{60} \rceil 5 + \lceil \tfrac{150}{70} \rceil 25 + \lceil \tfrac{150}{200} \rceil 25 \\
&= 30 + 30 + 15 + 75 + 25 = 175 \\
w_5^6(0) &= 30 + \lceil \tfrac{175}{50} \rceil 10 + \lceil \tfrac{175}{60} \rceil 5 + \lceil \tfrac{175}{70} \rceil 25 + \lceil \tfrac{175}{200} \rceil 25 \\
&= 30 + 40 + 15 + 75 + 25 = 185 \\
w_5^7(0) &= 30 + \lceil \tfrac{185}{50} \rceil 10 + \lceil \tfrac{185}{60} \rceil 5 + \lceil \tfrac{185}{70} \rceil 25 + \lceil \tfrac{185}{200} \rceil 25 \\
&= 30 + 40 + 20 + 75 + 25 = 190 \\
w_5^8(0) &= 30 + \lceil \tfrac{190}{50} \rceil 10 + \lceil \tfrac{190}{60} \rceil 5 + \lceil \tfrac{190}{70} \rceil 25 + \lceil \tfrac{190}{200} \rceil 25 \\
&= 30 + 40 + 20 + 75 + 25 = 190
\end{aligned}$$

The response time is $r_5 = w_5(0) = 190$.

Processor demand analysis

Under EDF, the auxiliary task set can be analysed by exisiting techniques without modification. Following Buttazzo [30] and George et al. [63], we use the following definitions:

$$\mathcal{D} = \{d \mid d = qT_i + D_i,\ d \le L,\ 0 \le i \le n,\ q \ge 0\}$$

$$h(d) = \sum_{D_i \le d} \left(1 + \left\lfloor \frac{d - D_i}{T_i} \right\rfloor\right) C_i$$

where $\mathcal{D}$ represents the absolute deadlines within the busy period, and $h(d)$ denotes the processor demand in the interval of length d following the critical instant.

For the auxiliary task, we have the busy period $L = 190$, which gives the following $\mathcal{D}$:

$$\begin{aligned}
\mathcal{D} &= \{d \mid d = kT_i + D_i,\ d \le 190,\ 1 \le i \le n,\ k \ge 0\} \\
&= \{d \mid d = 50k + 30,\ d \le 190,\ k \ge 0\} \cup \\
&\quad \{d \mid d = 60k + 100,\ d \le 190,\ k \ge 0\} \cup \\
&\quad \{d \mid d = 70k + 100,\ d \le 190,\ k \ge 0\} \cup \\
&\quad \{d \mid d = 200k + 100,\ d \le 190,\ k \ge 0\} \cup \\
&\quad \{d \mid d = 200k + 200,\ d \le 190,\ k \ge 0\} \\
&= \{30,\ 80,\ 130,\ 180\} \cup \{100,\ 160\} \cup \{100,\ 170\} \cup \{100\} \cup \{\} \\
&= \{30,\ 80,\ 100,\ 130,\ 160,\ 170,\ 180\}
\end{aligned}$$

Next, we compute $h(d)$ for each $d \in \mathcal{D}$:

$$\begin{aligned}
h(30) &= \textstyle\sum_{D_i \le 30} \left(1 + \left\lfloor \frac{30 - D_i}{T_i} \right\rfloor\right) C_i \\
&= \left(1 + \left\lfloor \frac{30 - D_1}{T_1} \right\rfloor\right) C_1 \ = \ \left(1 + \left\lfloor \frac{0}{50} \right\rfloor\right) 10 \ = \ 10
\end{aligned}$$

$$\begin{aligned}
h(80) &= \textstyle\sum_{D_i \le 80} \left(1 + \left\lfloor \frac{80 - D_i}{T_i} \right\rfloor\right) C_i \\
&= \left(1 + \left\lfloor \frac{80 - D_1}{T_1} \right\rfloor\right) C_1 \ = \ \left(1 + \left\lfloor \frac{50}{50} \right\rfloor\right) 10 \ = \ 20
\end{aligned}$$

$$\begin{aligned}
h(100) &= \textstyle\sum_{D_i \le 100} \left(1 + \left\lfloor \frac{100 - D_i}{T_i} \right\rfloor\right) C_i \\
&= \left(1 + \left\lfloor \frac{100 - D_1}{T_1} \right\rfloor\right) C_1 + \left(1 + \left\lfloor \frac{100 - D_2}{T_2} \right\rfloor\right) C_2 + \\
&\quad \left(1 + \left\lfloor \frac{100 - D_3}{T_3} \right\rfloor\right) C_3 + \left(1 + \left\lfloor \frac{100 - D_4}{T_4} \right\rfloor\right) C_4 \\
&= \left(1 + \left\lfloor \frac{70}{50} \right\rfloor\right) 10 + \left(1 + \left\lfloor \frac{0}{60} \right\rfloor\right) 5 + \\
&\quad \left(1 + \left\lfloor \frac{0}{70} \right\rfloor\right) 25 + \left(1 + \left\lfloor \frac{0}{200} \right\rfloor\right) 25 \\
&= 20 + 5 + 25 + 25 \ = \ 75
\end{aligned}$$

$$
\begin{aligned}
h(130) &= \textstyle\sum_{D_i \le 130} \left(1 + \left\lfloor \frac{130 - D_i}{T_i} \right\rfloor\right) C_i \\
&= \left(1 + \left\lfloor \frac{130 - D_1}{T_1} \right\rfloor\right) C_1 + \left(1 + \left\lfloor \frac{130 - D_2}{T_2} \right\rfloor\right) C_2 + \\
&\quad \left(1 + \left\lfloor \frac{130 - D_3}{T_3} \right\rfloor\right) C_3 + \left(1 + \left\lfloor \frac{130 - D_4}{T_4} \right\rfloor\right) C_4 \\
&= \left(1 + \left\lfloor \frac{100}{50} \right\rfloor\right) 10 + \left(1 + \left\lfloor \frac{30}{60} \right\rfloor\right) 5 + \\
&\quad \left(1 + \left\lfloor \frac{30}{70} \right\rfloor\right) 25 + \left(1 + \left\lfloor \frac{30}{200} \right\rfloor\right) 25 \\
&= 30 + 5 + 25 + 25 \;=\; 85
\end{aligned}
$$

$$
\begin{aligned}
h(160) &= \textstyle\sum_{D_i \le 160} \left(1 + \left\lfloor \frac{160 - D_i}{T_i} \right\rfloor\right) C_i \\
&= \left(1 + \left\lfloor \frac{160 - D_1}{T_1} \right\rfloor\right) C_1 + \left(1 + \left\lfloor \frac{160 - D_2}{T_2} \right\rfloor\right) C_2 + \\
&\quad \left(1 + \left\lfloor \frac{160 - D_3}{T_3} \right\rfloor\right) C_3 + \left(1 + \left\lfloor \frac{160 - D_4}{T_4} \right\rfloor\right) C_4 \\
&= \left(1 + \left\lfloor \frac{130}{50} \right\rfloor\right) 10 + \left(1 + \left\lfloor \frac{60}{60} \right\rfloor\right) 5 + \\
&\quad \left(1 + \left\lfloor \frac{60}{70} \right\rfloor\right) 25 + \left(1 + \left\lfloor \frac{60}{200} \right\rfloor\right) 25 \\
&= 30 + 10 + 25 + 25 \;=\; 90
\end{aligned}
$$

$$
\begin{aligned}
h(170) &= \textstyle\sum_{D_i \le 170} \left(1 + \left\lfloor \frac{170 - D_i}{T_i} \right\rfloor\right) C_i \\
&= \left(1 + \left\lfloor \frac{170 - D_1}{T_1} \right\rfloor\right) C_1 + \left(1 + \left\lfloor \frac{170 - D_2}{T_2} \right\rfloor\right) C_2 + \\
&\quad \left(1 + \left\lfloor \frac{170 - D_3}{T_3} \right\rfloor\right) C_3 + \left(1 + \left\lfloor \frac{170 - D_4}{T_4} \right\rfloor\right) C_4 \\
&= \left(1 + \left\lfloor \frac{140}{50} \right\rfloor\right) 10 + \left(1 + \left\lfloor \frac{70}{60} \right\rfloor\right) 5 + \\
&\quad \left(1 + \left\lfloor \frac{70}{70} \right\rfloor\right) 25 + \left(1 + \left\lfloor \frac{70}{200} \right\rfloor\right) 25 \\
&= 30 + 10 + 50 + 25 \;=\; 115
\end{aligned}
$$

$$
\begin{aligned}
h(180) &= \textstyle\sum_{D_i \le 180} \left(1 + \left\lfloor \frac{180 - D_i}{T_i} \right\rfloor\right) C_i \\
&= \left(1 + \left\lfloor \tfrac{180-D_1}{T_1} \right\rfloor\right) C_1 + \left(1 + \left\lfloor \tfrac{180-D_2}{T_2} \right\rfloor\right) C_2 + \\
&\quad \left(1 + \left\lfloor \tfrac{180-D_3}{T_3} \right\rfloor\right) C_3 + \left(1 + \left\lfloor \tfrac{180-D_4}{T_4} \right\rfloor\right) C_4 \\
&= \left(1 + \left\lfloor \tfrac{150}{50} \right\rfloor\right) 10 + \left(1 + \left\lfloor \tfrac{80}{60} \right\rfloor\right) 5 + \\
&\quad \left(1 + \left\lfloor \tfrac{80}{70} \right\rfloor\right) 25 + \left(1 + \left\lfloor \tfrac{80}{200} \right\rfloor\right) 25 \\
&= 40 + 10 + 50 + 25 \;=\; 125
\end{aligned}
$$

Finally, for each of the computed $h(d)$ values we check that $h(d) \le d$, :

$$
\begin{array}{llll}
h(30) = 10 & \le & 30 & \qquad h(160) = 90 \le 160 \\
h(80) = 20 & \le & 80 & \qquad h(170) = 115 \le 170 \\
h(100) = 75 & \le & 100 & \qquad h(180) = 125 \le 180 \\
h(130) = 85 & \le & 130 &
\end{array}
$$

Since all the above inequalities hold, the task set is schedulable under EDF.

Appendix D

Publication List

These publications have been (co-)authored by the author of this thesis.

Journal publications

- **The SAVE Approach to Component-Based Development of Vehicular Systems.** Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson and Massimo Tivoli. *Journal of Systems and Software*, vol 80, nr 5, p655–667. Elsevier, May 2007.

Theses

- **An Intuitive and Resource-Efficient Event Detection Algebra.** Jan Carlson. *Mälardalen University, Licentiate thesis No. 29.* ISBN 91-88834-49-2. June 2004.

Conference publications

- **Determining Maximum Stack Usage in Preemptive Shared Stack Systems.** Kaj Hänninen, Jukka Mäki-Turja, Markus Bohlin, Jan Carlson and Mikael Nolin. In *Proceedings of the 27th IEEE Real-Time Systems Symposium, Rio de Janeiro, Brazil.* December 2006.

- **Merging In-House Developed Software Systems – A Method for Exploring Alternatives.** Rikard Land, Jan Carlson, Ivica Crnkovic and Stig Larsson. In *Proceedings of the 2nd International Conference on Quality of Software Architecture, Västerås, Sweden.* June 2006.

- **An Event Detection Algebra for Reactive Systems.** Jan Carlson and Björn Lisper. In *Proceedings of the fourth ACM International Conference on Embedded Software (EMSOFT'04), Pisa, Italy.* September 2004.

- **Enhancing Time Triggered Scheduling with Value Based Overload Handling and Task Migration.** Jan Carlson, Tomas Lennvall and Gerhard Fohler. In *Proceedings of the 6th IEEE International Symposium on Object-oriented Real-time distributed Computing, Hakodate, Japan.* May 2003.

- **Value Based Overload Handling of Aperiodic Tasks in Offline Scheduled Real-Time Systems.** Jan Carlson, Tomas Lennvall and Gerhard Fohler. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (Work-in-progress Session), Delft, The Netherlands.* June 2001.

Workshop publications

- **Handling Subsystems using the SaveComp Component Technology.** Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, Mikael Nolin, Thomas Nolte, John Håkansson and Paul Pettersson. In *Proceedings of the 1st Workshop on Models and Analysis for Automotive Systems (WMAAS'06), Rio de Janeiro, Brazil.* December 2006.

- **SaveCCM: An Analysable Component Model for Real-Time Systems.** Jan Carlson, John Håkansson and Paul Pettersson. In *Proceedings of the 2nd International Workshop on Formal Aspects of Component Software (FACS05), Macao.* Elsevier, October 2005

- **An Event Algebra Extension of the Triggering Mechanism in a Component Model for Embedded Systems.** Jan Carlson and Mikael Åkerholm. In *Proceedings of the Workshop on Formal*

Foundations of Embedded Software and Component-Based Software Architectures (FESCA), Edinburgh, Scotland. April 2005.

- **An Interval-Based Algebra for Restricted Event Detection.** Jan Carlson and Björn Lisper, In *Proceedings of the First International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS 2003), Marseille, France.* September 2003.

Appendix E

Notation List

Notation	Description	Page
;	sequence operator	52
$\oplus$	instance constructor function	53
$[\![\ ]\!]^{\mathcal{I}}$	algebra semantics	55
$\|X\|$	the maximum size of a set variable X	73
$+$	conjunction operator	52
$-$	negation operator	52
$\vee$	disjunction operator	52
ε	a non-occurrence	65
$\equiv$	expression equivalence	59
$\mathbf{0}$	empty event	61
$\mathcal{A}(i)$	the event stream detected for E^i	67
$acorr$	output correctness property	70
a_i	auxiliary instance variable	65
A_τ	temporal restriction	52

(continued on next page)

(continued from previous page)

Notation	Description	Page
C_i	worst case execution time of task t_i	85
compsize	the size of a composite instance	78
con	conjunction operator semantics function	55
D	event instance domain	53
$\mathcal{D}$	absolute deadlines within the busy period	98
D_i	relative deadline of task t_i	85
dis	disjunction operator semantics function	55
E	the expression to be detected	65
e, e'	temporary instance variable	65
E^i	the ith subexpression of E	65
E_i	event expression of task t_i	85
empty	empty stream interval	123
end	end time of an instance	53
$ep(i)$	tasks of the same priority as t_i	94
Γ	original taskset	90
Γ^{aux}	auxiliary taskset	90
$h(d)$	processor demand	98
$hp(i)$	tasks of higher priority than t_i	94
$\mathcal{I}$	interpretation	54
i	instance size (analysis algorithm)	77
L	busy period	98
l_i	persistent instance variable	65
$L(P_i)$	level P_i busy period	92

(continued on next page)

(continued from previous page)

Notation	Description	Page
m	abstract memory usage (analysis algorithm)	76
m	the number of subexpressions in E	65
mint(A)	minimum interarrival time of A	88
neg	negation operator semantics function	55
$\mathcal{P}$	primitive events (identifiers)	52
pcorr	S_i correctness property	68
P_i	priority of task t_i	92
prim(E)	the primitive events of E	89
primsize	maximum instance size of a primitive event	78
Q'	temporary instance set variable	65
Q_i	persistent instance set variable	65
res	restriction policy	57
r_i	persistent instance variable	65
r_i	worst case response time of t_i	94
s	size of S_i (analysis algorithm)	77
seq	sequence operator semantics function	55
S_i	auxiliary time set variable	65
sn	is S_i needed (analysis algorithm)	77
start	start time of an instance	53
state	state information correctness property	68
subexp(A)	the number of subexpressions in A	73
$\mathcal{T}$	temporal domain	51
t	abstract execution time (analysis algorithm)	76

(continued on next page)

(continued from previous page)

Notation	Description	Page
t	temporary time variable	65
term(E)	the terminating primitive events of E	89
T_i	period of task t_i	85
t_i	persistent time variable	65
tim	temporal restriction semantics function	55
type(p)	the type of port p	107
U	processor utilisation	97
unionsize	disjunction instance size	78
valid	"pointwise restriction predicate"	69
ω	maximum instance size	74
wcet(E)	the WCET associated with the detection of E	89
$w_i(q)$	latest finishing time of the qth instance of t_i	94

Bibliography

[1] Jonathon Abbott, Jim Bell, Andrew Clark, Olivier De Vel, and George Mohay. Automated recognition of event scenarios for digital forensics. In *SAC'06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 293–300. ACM Press, 2006.

[2] AbsInt company homepage. Accessed Mar 20, 2007. `http://www.absint.com`.

[3] Raman Adaikkalavan and Sharma Chakravarthy. Formalization and detection of events using interval-based semantics. In Jayant R. Haritsa and T. M. Vijayaraman, editors, *Advances in Data Management 2005, Proceedings of the Eleventh International Conference on Management of Data, January 6, 7, and 8, 2005, Goa, India*, pages 58–69. Computer Society of India, 2005.

[4] Raman Adaikkalavan and Sharma Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. *Data & Knowledge Engineering*, 59(1):139–165, 2006.

[5] Jose Aguilar, Kouamana Bousson, Christophe Dousson, Malik Ghallab, Antony Guasch, Rob Milne, Charlie Nicol, Jose Quevedo, and Louise Travé-Massuyès. TIGER: Real-time situation assessment of dynamic systems. *Intelligent Systems Engineering*, pages 103–124, October 1994.

[6] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The SAVE approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, 2007.

[7] Mikael Åkerholm, Anders Möller, Hans Hansson, and Mikael Nolin. Towards a dependable component technology for embedded system applications. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005)*. IEEE, January 2005.

[8] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[9] James F. Allen and George Ferguson. Actions and events in interval temporal logic. *Journal of Logic and Computation*, 4(5):531–579, October 1994.

[10] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[11] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 74–106, Berlin, Germany, June 1992. Springer.

[12] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *Proc. of 1st International Workshop on Formal Modeling and Analysis of Timed Systems*, Lecture Notes in Computer Science. Springer, 2003.

[13] James H. Anderson, Vasile Bud, and UmaMaheswari C. Devi. An edf-based scheduling algorithm for multiprocessor soft real-time systems. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 199–208. IEEE Computer Society, 2005.

[14] Edward Angel. *Interactive Computer Graphics*. Addison-Wesley, 2002.

[15] Neil C. Audsley, Alan Burns, Robert I. Davis, Ken W. Tindell, and Andy J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Syst.*, 8(2-3):173–198, 1995.

[16] James Bailey and Szabolcs Mikulás. Expressiveness issues and decision problems for active database event queries. In *Database Theory - ICDT 2001, 8th International Conference*, volume 1973 of *Lecture Notes in Computer Science*, pages 68–82, London, UK, 4–6 January 2001. Springer.

[17] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[18] Sanjoy K. Baruah, Louis E. Rosier, and Rodney R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, 1990.

[19] Pierfrancesco Bellini, Riccardo Mattolini, and Paolo Nesi. Temporal logics for real-time system specification. *ACM Computing Surveys*, 32(1):12–42, March 2000.

[20] Martin Bernauer, Gerti Kappel, and Gerhard Kramler. Composite events for XML. In *Proceedings of WWW2004*, New York, USA, May 17–22 2004. ACM.

[21] Mikael Berndtsson and Jörgen Hansson. Issues in active real-time databases. In *Active and Real-Time Database Systems*, pages 142–157, 1995.

[22] Karthikeyan Bhargavan and Carl A. Gunter. Network event recognition. *Form. Methods Syst. Des.*, 27(3):213–251, 2005.

[23] Karthikeyan Bhargavan and Carl A. Gunter. Network event recognition. *Formal Methods in System Design*, 27(3):213–251, 2005.

[24] Bound-T homepage. Accessed Mar 20, 2007. `http://www.tidorum.fi/bound-t/`.

[25] T. Bowen, D. Chee, M. Segal, R. Sekar, T. Shanbhag, and P. Uppuluri. Building survivable systems: An integrated approach based on intrusion detection and damage containment. In *Proceedings of the DARPA Information Survivability Conference and Exposition*

(DISCEX 2000), pages 1084–1099. IEEE Computer Society Press, January 2000.

[26] Don Box. *Essential COM*. Addison-Wesley, 1997.

[27] Alan Burns, Ken Tindell, and Andy Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Trans. Softw. Eng.*, 21(5):475–480, 1995.

[28] Alan Burns and Andy Wellings. *Real-time systems and their programming languages*. Addison-Wesley, 1990.

[29] Stanley Burris and Hanamantagouda P. Sankappanavar. *A Course in Universal Algebra*. Springer-Verlag, New York, 1981.

[30] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[31] Giorgio C. Buttazzo and John A. Stankovic. Adding robustness in dynamic preemptive scheduling. In Donald Fussell and Miroslaw Malek, editors, *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*. Kluwer Academic Publishers, 1995.

[32] Jan Carlson. An intuitive and resource-efficient event detection algebra. Licentiate thesis No. 29, June 2004. Mälardalen University, Sweden.

[33] Jan Carlson, John Håkansson, and Paul Pettersson. SaveCCM: An analysable component model for real-time systems. In Z. Liu and L. Barbosa, editors, *Proceedings of the 2nd Workshop on Formal Aspects of Components Software (FACS 2005)*, volume 160 of *Electronic Notes in Theoretical Computer Science*, pages 127–140. Elsevier, 2006.

[34] Sharma Chakravarthy, Vidhya Krishnaprasad, Eman Anwar, and Seung-Kyum Kim. Composite events for active databases: Semantics, contexts and detection. In *20th International Conference on Very Large Data Bases*, pages 606–617, Santiago, Chile, 12–15 September 1994. Morgan Kaufmann Publishers.

[35] Sharma Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[36] Sharma Chakravathy and Jae Dong Yang. A recursion-based framework for detecting composite events in active databases, January 1998.

[37] Houssine Chetto, Maryline Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194, 1990.

[38] Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.

[39] Georgio Chrysanthakopoulos and Satnam Singh. An asynchronous messaging library for C#. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, October 2005.

[40] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[41] James Conard, Patrick Dengler, Brian Francis, Jay Glynn, Burton Harvey, Billy Hollis, Rama Ramachandran, John Schenken, Scott Short, and Chris Ullman. *Introducing .NET*. Wrox Press, 2001.

[42] Ivica Crnkovic. Component-based approach for embedded systems. In *Ninth International Workshop on Component-Oriented Programming*, June 2004.

[43] Ivica Crnkovic. Component-based approach for embedded systems. In *Ninth International Workshop on Component-Oriented Programming*, June 2004.

[44] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems*. Artech House, 2002.

[45] Michael L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.

[46] Radu Dobrin. *Combining Off-line Schedule Construction and Fixed Priority Scheduling in Real-Time Computer Systems*. PhD thesis, Mälardalen University, September 2005.

[47] C. Dousson, P. Gaborit, and M. Ghallab. Situation recognition: Representation and algorithms. In *Proceedings of 13th International Joint Conference on Artificial Intelligence*, pages 166–172, Chambery, France, 1993.

[48] Christophe Dousson. Alarm driven supervision for telecommunication networks: II- On-line chronicle recognition. *Annals of Telecommunications*, pages 501–508, October 1996. CNET, France Telecom.

[49] Dan Egnor. liboop home page. Accessed Jan 12, 2007. `http://liboop.ofb.net/`.

[50] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[51] Pascal Fenkam. *A Systematic Approach to the Development of Event-Based Applications*. PhD thesis, Technical University of Vienna, October 2003.

[52] David Flanagan. *Java in a Nutshell (2nd ed.)*. O'Reilly & Associates, Inc., 1997.

[53] Gerhard Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *IEEE Real-Time Systems Symposium*, December 1995.

[54] P. Frohlich, W. Nejdl, K. Jobmann, and H. Wietgrefe. Model-based alarm correlation in cellular phone networks, 1997.

[55] Antony Galton and Juan Carlos Augusto. Two approaches to event definition. In *Proc. of Database and Expert Systems Applications 13th Int. Conference (DEXA'02)*, volume 2453 of *Lecture Notes in Computer Science*. Springer-Verlag, September 2002.

[56] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[57] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Proc. 1st Intl. Workshop on Rules in Database Systems (RIDS)*, Edinburgh, UK, September 1993. Springer-Verlag.

[58] S. Gatziu and K. R. Dittrich. Detecting composite events in active database systems using petri nets. In *Research Issues in Data Engineering (RIDE '94)*, pages 2–9, Los Alamitos, Ca., USA, February 1994. IEEE Computer Society Press.

[59] N. Gehani, H. V. Jagadish, and O. Shmueli. COMPOSE: A system for composite specification and detection. In *Advanced Database Systems*, volume 759 of *Lecture Notes in Computer Science*. Springer, 1993.

[60] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of the 17th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Barcelona*, 1991.

[61] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an Active Object–Oriented Database. In *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pages 81–90, 1992.

[62] Narain H. Gehani and Daniel F. Lieuwen. Ode triggers: Monitoring the stock market. *Software — Practice and Experience*, 27(8):905–927, 1997.

[63] Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Technical Report RR-2966, Institut National de Recherche et Informatique et en Automatique, 1996.

[64] Rodolfo Gómez and Juan Carlos Augusto. Durative events in active databases. In *ICEIS 2004, Proceedings of the 6th International Conference on Enterprise Information Systems*, pages 306–311, Porto, Portugal, April 2004.

[65] R. E. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the READY event notification service. In P. Dasgupta, editor, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, Middleware Workshop*, Austin, TX, USA, May 1999.

[66] Joshua Haines, Dorene Kewley Ryder, Laura Tinnel, and Stephen Taylor. Validation of sensor alert correlators. *IEEE Security and Privacy*, 1(1):46–56, 2003.

[67] Jeri R. Hanly and Elliot B. Koffman. *Problem Solving and Program Design in C*. Addison-Wesley, 1995.

[68] Jörgen Hansson and Mikael Berndtsson. Active real-time database systems. In *Active Rules in Database Systems*, pages 405–426. Springer, 1999.

[69] Lilian Harada and Yuuji Hotta. Order checking in a CPOE using event analyzer. In *CIKM'05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 549–555. ACM Press, 2005.

[70] Lilian Harada, Yuuji Hotta, and Tadashi Ohmori. Detection of sequential patterns of events for supporting business intelligence solutions. In *IDEAS '04: Proceedings of the International Database Engineering and Applications Symposium (IDEAS'04)*, pages 475–479, Washington, DC, USA, 2004. IEEE Computer Society.

[71] Michael González Harbour, Mark H. Klein, and John P. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priority. In Robert Werner, editor, *Proceedings of the Real-Time Systems Symposium - 1991*, pages 116–128. IEEE Computer Society Press, December 1991.

[72] Richard Hayton, Jean Bacon, John Bates, and Ken Moody. Using events to build large scale distributed applications. In *Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 9–16, New York, NY, USA, 1996. ACM Press.

[73] George T. Heineman and William T. Councill, editors. *Component-based software engineering: Putting the pieces together*. Addison-Wesley, 2001.

[74] David M. Hilbert and David F. Redmiles. Agents for collecting application usage data over the Internet. In Katia P. Sycara and Michael Wooldridge, editors, *Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98)*, pages 149–156. ACM Press, 9–13, 1998.

[75] A. Hinze and A. Voisard. A flexible parameter-dependent algebra for event notification services. Technical Report TR-B-02-10, Freie Universität Berlin, 2002.

[76] A. Hinze and A. Voisard. A parameterized algebra for event notification services. In *Proceedings of the 9th International Symposium on Temporal Representation and Reasoning (TIME 2002)*, Manchester, UK, July 2002. Springer-Verlag.

[77] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, 1979.

[78] G. Jakobson and M. Weissman. Alarm correlation. *IEEE Network Magazine*, 6(7):52–59, 1993.

[79] K. Jeffay and D. L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In Susan Davidson and Insup Lee, editors, *Proceedings of the Real-Time Systems Symposium*, pages 212–221. IEEE Computer Society Press, December 1993.

[80] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use, vol 1*. EATCS Monographs in Computer Science. Springer-Verlag, 1992.

[81] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal (British Computer Society)*, 29(5):390–395, Oct 1986.

[82] Iluju Kiringa. Specifying event logics for active databases. In *Proceedings of the 2002 International Workshop on Description Logics (DL2002), Toulouse, France*, volume 53 of *CEUR Workshop Proceedings*, 2002.

[83] Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. *A Practitioner's Handbook for Real-Time Analysis : Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[84] Hermann Kopetz. Should responsive systems be event-triggered or time-triggered? *IEICE Transactions on Information and Systems*, E76-D(11):1325–1332, 1993.

[85] R. Kowalski. Database updates in the event calculus. *The Journal of Logic Programming*, 12:121, January 1992.

[86] R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[87] S. Krakowiak. What is middleware, 2003. `http://middleware.objectweb.org`.

[88] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[89] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, Dec 1982.

[90] Guoli Li and Hans-Arno Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *6th International Middleware Conference, Grenoble, France*, volume 3790 of *Lecture Notes in Computer Science*, pages 249–269. Springer, 2005.

[91] C. Liebig, B. Boesling, and A. Buchmann. A notification service for next-generation it systems in air traffic control. In *GI-Workshop: Multicast-Protokolle und Anwendungen*, Braunschweig, Germany, May 1999.

[92] C. Liebig, M. Cilia, and A. Buchmann. Event composition in time-dependent distributed systems. In *Proceedings of the 4th Intl. Conference on Cooperative Information Systems (CoopIS '99)*, September 1999.

[93] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[94] G. Liu, A. Mok, and P. Konana. A unified approach for specifying timing constraints and composite events in active real-time database systems. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*, pages 199–209, Washington - Brussels - Tokyo, June 1998. IEEE.

[95] G. Liu, A. K. Mok, and E. J. Yang. Composite events for network event correlation. In *Proceedings of the 6th IFIP/IEEE International Symposium on Integrated Network Management*, pages 247–260. IEEE, 1999.

[96] C. Douglas Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, 1992.

[97] Chaoying Ma and Jean Bacon. COBEA: A CORBA-based event architecture. In *Proceedings of the USENIX Conference on Object-Oriented Technologies and Systems*, pages 117–131, June 1998.

[98] Masoud Mansouri-Samani and Morris Sloman. GEM: A generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, June 1997.

[99] K. Meinke and J. V. Tucker. Universal algebra. In *Handbook of Logic in Computer Science*, volume 1, pages 189–368. Oxford University Press, 1992.

[100] Jonas Mellin. *Resource-Predictable and Efficient Monitoring of Events*. PhD thesis, Department of Computer and Information Science, Linköping University, June 2004. Dissertation No 876.

[101] Jonas Mellin and Sten F. Andler. A formalized schema for event composition. In *Proc. 8th Int. Conf on Real-Time Computing Systems and Applications (RTCSA 2002)*, pages 201–210, Tokyo, Japan, 18–20 March 2002.

[102] Microsoft Corporation. Microsoft Robotics Studio. Accessed Mar 9, 2007. `http://msdn.microsoft.com/robotics/`.

[103] A. Mok and G. Liu. Early detection of timing constraint violation at runtime. In *The 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 176–186, Washington - Brussels - Tokyo, December 1997. IEEE.

[104] A. Mok and G. Liu. Efficient run-time monitoring of timing constraints. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 252–262, Washington - Brussels - Tokyo, June 1997. IEEE.

[105] Aloysius K. Mok, Prabhudev Konana, Guangtian Liu, Chan-Gun Lee, and Honguk Woo. Specifying timing constraints and composite events: An application in the design of electronic brokerages. *IEEE Transactions on Software Engineering*, 30(12):841–858, 2004.

[106] Anders Möller, Jakob Engblom, and Mikael Nolin. Developing and testing distributed CAN-based real-time control-systems using a single PC. In *10th International CAN Conference, Roma, Italy*. CAN in Automation, March 2005.

[107] Anders Möller, Joakim Fröberg, and Mikael Nolin. Industrial requirements on component technologies for embedded systems. In *International Symposium on Component-based Software Engineering (CBSE7)*, Edinburgh, Scotland, May 2004. Springer Verlag.

[108] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.

[109] Iakovos Motakis and Carlo Zaniolo. Formal semantics for composite temporal events in active database rules. *Journal of Systems Integration*, 7(3/4):291–325, 1997.

[110] Mozilla Developer Center. JavaScript. Accessed Mar 13, 2007. `http://developer.mozilla.org/en/docs/JavaScript`.

[111] Prasad Naldurg, Koushik Sen, and Prasanna Thati. A temporal logic based framework for intrusion detection. In David de Frutos-Escrig and Manuel Núñez, editors, *Formal Techniques for Networked and Distributed Systems (FORTE 2004), 24th IFIP WG 6.1 International Conference, Madrid Spain*, volume 3235 of *Lecture Notes in Computer Science*, pages 359–376. Springer, 2004.

[112] O. Nierstrass, G. Arevalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Zeidler, T. Genssler, and R. van den Born. A component model for field devices. In *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment*, June 2002.

[113] Object Management Group. Event Service Specification, May 1997. `http://www.omg.org`.

[114] Object Management Group. Minimum CORBA Specification, v1.0, August 2002. `http://www.omg.org`.

[115] Object Management Group. Real-Time CORBA Specification, v1.2, January 2005. `http://www.omg.org`.

[116] Object Management Group. CORBA Component Model Specification, v4.0, April 2006. `http://www.omg.org`.

[117] Balaji Padmanabhan and Alexander Tuzhilin. On characterization and discovery of minimal unexpected patterns in rule discovery. *IEEE Trans. Knowl. Data Eng.*, 18(2):202–216, 2006.

[118] Satyaraj Pantham. *Pure JFC Swing*. Sams, 1999.

[119] N. W. Paton, J. Campin, A. A. A. Fernandes, and M. H. Williams. Formal specification of active database functionality: A survey. In T. Sellis, editor, *Proceedings of the 2nd International Workshop on Rules in Database Systems*, volume 985, pages 21–35. Springer, 1995.

[120] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999.

[121] Monika Pfau-Wagenbauer and Wolfgang Nejdl. Integrating model-based and heuristic features in a real-time expert system. *IEEE Expert: Intelligent Systems and Their Applications*, 8(4):12–18, 1993.

[122] Peter R. Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, University of Cambridge, February 2004.

[123] Peter R. Pietzuch, Brian Shand, and Jean Bacon. Composite event detection as a generic middleware extension. *IEEE Network*, 18(1):44–55, 2004.

[124] Niels Provos. libevent – an event notification library. Accessed Jan 12, 2007. Last modified Apr 7, 2004. `http://monkey.org/~provos/libevent/`.

[125] Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. In John A. Stankovic and Krithi Ramamritham, editors, *Advances in Real-Time Systems*, pages 322–339. IEEE Computer Society Press, 1993.

[126] Quadros Systems, Inc. RTXC Quadros Overview. Accessed May 14, 2007. http://www.quadros.com/products/operating-systems/rtxc-quadros/.

[127] Parameswaran Ramanathan, Kang G. Shin, and Ricky W. Butler. Fault-tolerant clock synchronization in distributed systems. *Computer*, 23(10):33–42, 1990.

[128] Ismael Ripoll, Alfons Crespo, and Aloysius K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11(1):19–39, 1996.

[129] Ismael Ripoll, Alfons Crespo, and Aloysius K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Syst.*, 11(1):19–39, 1996.

[130] Claudia L. Roncancio. Toward duration-based, constrained and dynamic event types. In Sten Andler and Jörgen Hansson, editors, *Proceedings of the 2nd International Workshop on Active, Real-Time, and Temporal Database Systems*, volume 1553 of *Lecture Notes in Computer Science*, pages 176–193. Springer, 1998.

[131] C. Sánchez, S. Sankaranarayanan, H. Sipma, T. Zhang, D. Dill, and Z. Manna. Event correlation: Language and semantics. In *Embedded Software, Third International Conference, EMSOFT 2003*, volume 2855 of *Lecture Notes in Computer Science*, pages 323–33. Springer, 2003.

[132] César Sánchez, Henny B. Sipma, Matteo Slanina, and Zohar Manna. Final semantics for Event-Pattern Reactive Programs. In *First International Conference in Algebra and Coalgebra in Computer Science (CALCO'05)*, volume 3629 of *LNCS*, pages 364–378. Springer-Verlag, September 2005.

[133] César Sánchez, Matteo Slanina, Henny B. Sipma, and Zohar Manna. Expressive completeness of an event-pattern reactive programming language. In Farn Wang, editor, *FORTE*, volume 3731 of *Lecture Notes in Computer Science*, pages 529–532. Springer, 2005.

[134] SAVE project homepage. Accessed Mar 31, 2007. http://www.mrtc.mdh.se/SAVE/.

[135] Herbert Schildt. *C#: A Beginners Guide*. McGraw-Hill Companies, 2001.

[136] Scarlet Schwiderski. *Monitoring the behaviour of distributed systems*. PhD thesis, University of Cambridge, April 1996.

[137] Koushik Sen, Grigore Roşu, and Gul Agha. Generating optimal linear temporal logic monitors by coinduction. In *8th Asian Computing Science Conference (ASIAN'03)*, volume 2896 of *Lecture Notes in Computer Science*, pages 260–75. Springer, 2003.

[138] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE ransaction on Computers*, 39(9):1175–1185, 1990.

[139] A. Prasad Sistla and Ouri Wolfson. Temporal triggers in active databases. *Knowledge and Data Engineering*, 7(3):471–486, 1995.

[140] Rajendran M. Sivasankaran, John A. Stankovic, Donald F. Towsley, Bhaskar Purimetla, and Krithi Ramamritham. Priority assignment in real-time active databases. *VLDB Journal: Very Large Data Bases*, 5(1):19–34, 1996.

[141] Marco Spuri. Analysis of deadline scheduled real-time systems. Technical Report RR-2772, Institut National de Recherche et Informatique et en Automatique, 1996.

[142] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *PODS '04: Proceedings of the 23rd symposium on Principles of database systems*, pages 263–274. ACM Press, 2004.

[143] Judith A. Stafford and Kurt Wallnau. Component composition and integration. In Ivica Crnkovic and Magnus Larsson, editors, *Building reliable component-based software systems*, pages 179–191. Artech House Publishers, 2002.

[144] Neil R. Storey. *Safety Critical Computer Systems*. Addison-Wesley, 1996.

[145] Sun Microsystems. JavaBeans, 8 August 1997. `http://java.sun.com/products/javabeans/docs/spec.html`.

[146] Andrew S. Tanenbaum. *Distributed operating systems*. Prentice-Hall, Inc., 1995.

[147] Ken W. Tindell, Alan Burns, and Andy J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[148] Frank Vahid and Tony Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. Wiley, 2001.

[149] Rob van Ommering, Frank van der Linden, Kramer Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, March 2000.

[150] Moshe Y. Vardi. Linear vs. branching time: A complexity-theoretic perspective. In Vaughan Pratt, editor, *Proceedings of the Thirteenth Annual IEEE Symp. on Logic in Computer Science, LICS 1998*, pages 394–405. IEEE Computer Society Press, June 1998.

[151] Shengquan Wang, Sangig Rho, Zhibin Mai, Riccardo Bettati, and Wei Zhao. Real-time component-based systems. In *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 428–437. IEEE Computer Society, 2005.

[152] Brent Welch, Ken Jones, and Jeffrey Hobbs. *Practical Programming in Tcl and Tk*. Prentice Hall, 2003.

[153] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem – Overview of methods and survey of tools. Technical report, Mälardalen Real-Time Research Centre, Mälardalen University, March 2007.

[154] Worst case execution times (WCET) project homepage. Accessed Mar 20, 2007. `http://www.mrtc.mdh.se/projects/wcet`.

[155] Peter Wright. *Beginning Visual Basic 5*. Wrox Press Ltd., 1996.

[156] Jia Xu and David Lorge Parnas. Priority scheduling versus pre-run-time scheduling. *Real-Time Syst.*, 18(1):7–23, 2000.

[157] Eiko Yoneki. *ECCO: Data centric asynchronous communication.* PhD thesis, University of Cambridge, Computer Laboratory, December 2006.

[158] R. Zhang and E. Unger. Event specification and detection. Technical Report TR CS-96-8, Department of Computing and Information Sciences, Kansas State University, June 1996.

[159] Q. Zheng and K. G. Shin. On the ability of establishing real-time channels in point-to-point packet-switched networks. *IEEE Transactions on Communications*, 42:1096–1105, February 1994.

[160] Dong Zhu and Adarshpal S. Sethi. SEL, A new event pattern specification language for event correlation. In *Proceeding of the 10th International Conference on Computer Communications and Networks (ICCCN)*, pages 586–589. IEEE, October 2001.

[161] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *Proceedings of the 15th International Conference on Data Engineering*, pages 392–399. IEEE Computer Society Press, 1999.

Index