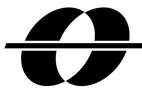
Mälardalen University Licentiate Thesis
No.29

An Intuitive and Resource-Efficient Event Detection Algebra

Jan Carlson

June 2004



MÄLARDALEN UNIVERSITY

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

Copyright © Jan Carlson, 2004
ISSN 1651-9256
ISBN 91-88834-49-2
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

In reactive systems, execution is driven by external events to which the system should respond with appropriate actions. Such events can be simple, but systems are often supposed to react to sophisticated situations involving a number of simpel events occurring in accordance with some pattern. A systematic approach to handle this type of systems is to separate the mechanism for detecting composite events from the rest of the application logic. A detection mechanism listens for simple event occurrences and notifies the application when one of the complex event patterns of interest occur. The event detection mechanism can for example be based on an event algebra, i.e., expressions that correspond to the event patterns of interest are built from simple events and operators from the algebra.

This thesis presents a novel event algebra with two important characteristics: It complies with algebraic laws that intuitively ought to hold for the operators of the algebra, and for a large class of expressions the detection can be correctly performed with limited resources in terms of memory and time. In addition to the declarative algebra semantics, we present an imperative detection algorithm and show that it correctly implements the algebra. This algorithm is analysed with respect to memory requirements and execution time complexity. To increase the efficiency of the algebra, we also present a semantic-preserving transformation scheme by which many expressions can be transformed to meet criteria under which limited resource requirements are guaranteed. Finally, we present a prototype implementation that combines the algebra with the event system in Java.

Why can't you just forget about algebra
it's all about you now
And all your talk of logic and formula
could never help you now
(not anymore)

A Camp – Algebra

Preface

A thesis should, tradition bids, start off by naming those
the author feels indebted to somehow, so here it goes:
Björn Lisper my advisor, first, for supervising me
and then just about everyone who works at IDt

For lengthy coffee breaks and talks on subjects quite diverse
in order based on syllables, to fit this freakin' verse:
Andreas, Markus, Waldemar, and Markus (well, they're two)
Nerina, Johan, Xavier and Lars, to name a few

At last the most important ones to mention in this list,
without your constant strong support no thesis would exist:
My fiancée Marina and of course my daughter Nell,
my parents and my siblings should be mentioned here as well

This list is far from finished, there are many still to add
If you have not been mentioned yet, you shouldn't feel too sad
Take comfort in the fact that I, because of lack of time,
did not forget you, only failed to find a proper rhyme.

Jan Carlson
Västerås, May 2004

Contents

1 Introduction **1**
1.1 Problem Formulation 3
1.2 The Approach 4
1.3 Related Publications 5
1.4 Contributions 6
1.5 Organisation 6

2 Event Detection **9**
2.1 Single Point or Interval Semantics 11
2.2 Event Contexts 12

3 The Event Algebra **15**
3.1 Declarative Semantics 15
3.1.1 Primitive Events 15
3.1.2 Composite Events 17
3.1.3 Semantics 19
3.2 Properties 23
3.2.1 Algebraic Laws 23
3.2.2 Impact from the Restriction Policy on the Laws . . 30
3.2.3 Properties of the Restriction Policy 32

4 An Event Detection Algorithm **37**
4.1 The Algorithm 37
4.1.1 Disjunction 40
4.1.2 Conjunction 41
4.1.3 Negation 43
4.1.4 Sequence 44
4.1.5 Temporal Restriction 45

4.1.6 Putting it all Together 46
4.2 Resource Requirements 46
4.2.1 Memory Complexity 46
4.2.2 Time Complexity 52

5 Expression Transformation 55
5.1 The Transformation Algorithm 55
5.2 Experiments 61
5.2.1 Experiment 1 61
5.2.2 Experiment 2 63

6 Prototype Implementation 65
6.1 Design Decisions 66
6.2 System Description 66
6.3 Using the Prototype 70

7 Related Work 73
7.1 Active Databases 73
7.2 Event Monitoring in Real-Time Systems 75
7.3 Event Notification Services 76
7.4 Knowledge Representation 77

8 Conclusions 79
8.1 Future Work 80
8.1.1 Evaluating the Algebra 80
8.1.2 Delay Operator 80
8.1.3 Dealing with Values 81
8.1.4 Implementing the Algebra 82

Bibliography 85

A Experimental Results 91
A.1 Experiment 1 91
A.2 Experiment 2 92

List of Figures

1.1 Integrated and separated detection of composite events. . 2

2.1 Comparison between single point semantics and interval semantics. 11
2.2 Comparison of three event contexts for the sequence operator. 14

3.1 Graphical representation of Example 3.3. 20
3.2 An event stream with two valid restrictions. 22
3.3 Graphical representation of Example 3.7. 35
3.4 Graphical representation of Example 3.8. 36

4.1 The event detection algorithm. 39
4.2 Statically simplified algorithm for detecting (T∨P)−B. . . 40
4.3 Detecting A;B_4 with bounded memory. 47
4.4 Algorithm for $E^i = E^j;E^k$ when $E^k \equiv E^k_{\tau'}$. 48
4.5 An example of how instances of A are stored in q_i and l_i during the detection of A;B_4. 53
4.6 Improved algorithm for $E^i = E^j;E^k$ when $E^k \equiv E^k_{\tau'}$. . . . 54

5.1 The recursive transformation function. 56
5.2 Transformation of P_2;(B∨T). 59
5.3 Transformation of $(\mathsf{B;B})_2$−(P;(P+T)). 60

6.1 Class diagram for the event algebra package. 67
6.2 Object diagram depicting how an expression is represented. 68
6.3 Sequence diagram depicting the communication within an expression. 70
6.4 Code for creating the expression $(\mathsf{A;(A\vee B)})_{5000}$. 71

A.1 Results for Experiment 1a. 93
A.2 Results for Experiment 1b. 94
A.3 Results for Experiment 2a with $T = 100$. 95
A.4 Results for Experiment 2a with $T = 300$. 96
A.5 Results for Experiment 2b with $T = 100$. 97
A.6 Results for Experiment 2b with $T = 300$. 98

List of Tables

2.1 Informal description of the algebra operators. 10

5.1 Summary of Experiment 1a. 62
5.2 Summary of Experiment 1b. 63
5.3 Summary of Experiment 2a. 64
5.4 Summary of Experiment 2b. 64

Chapter 1

Introduction

In a *reactive* system, execution is driven by a stream of external events to which the system should react with appropriate responses. A wide range of applications fall under this category, including active databases, systems for monitoring network traffic, electronic stock brokers, and many real-time and embedded systems.

For many reactive systems, the desired behaviour can be seen as reactions 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. This separation of concerns facilitates design and analysis of reactive systems, as detection of complex events can be given a formal semantics independent from the application in which it is used, and the remaining part of the system is free from auxiliary rules and information about partially completed patterns.

The event detection part reacts to the simple events of the system, referred to as *primitive events*, and detects the occurrences of *composite events* representing the complex event patterns of interest. In the rest of the system, these composite events are used to trigger specific actions in the same way as the primitive events.

Example 1.1. Consider a system with primitive events including a button B, a pressure alarm P and a temperature alarm T, where one desired reaction is that the system should perform the action A when the button is pressed twice within two seconds, unless either of the alarms occurs in between. This can be achieved by a set of rules that specify reactions to

the three primitive events, so that the combined behaviour implements the desired reaction. Alternatively, a separate detection mechanism can be used to define a composite event E that corresponds to the described situation, with a single rule stating that an occurrence of E should trigger the action A. The two approaches are illustrated by Figure 1.1. ◇

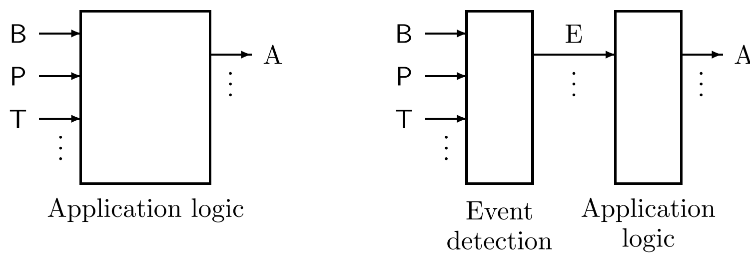


Figure 1.1: Integrated and separated detection of composite events.

A reactive systems is classified as a *real-time system* if it has temporal, as well as logical, constraints on the expected system behaviour. For this type of systems, correctness is defined as the ability to produce a correct result at the correct time. In a *hard* real-time system, a violated temporal constraint is considered a serious error, while systems classified as *soft* would consider it a performance degradation.

To establish the correctness of a hard real-time system, one must be able to show that no temporal constraints are violated, even in a worst case scenario. This is especially important in safety critical applications where a single constraint violation might cause serious damage. Ensuring timeliness requires that the resource requirements for all parts of the system, or at least safe approximations thereof, are known. If a separate detection mechanism is used, it must be possible to derive a bound on the memory required to correctly detect a given complex event pattern, as well as the worst case execution time of the detection mechanism for that pattern.

The mechanism to detect complex event patterns can be based on an *event algebra*, i.e., expressions that correspond to the event patterns of interest are built from simple events and operators from the algebra. Event algebras have been used in a variety of reactive system domains,

in particular for active databases but also in areas such as real-time systems and middleware platforms. It is desirable that an event algebra for reactive systems meets the following criteria:

- Sufficient expressiveness: The algebra should be rich enough to express many different types of composite events that might be of interest to the targeted type of systems.

- Formal semantics: A formal definition reduces ambiguity and facilitates reasoning about the algebra or a system that utilises it. In particular, formal reasoning about the system behaviour requires formal semantics.

- Intuitive operators: The usability of the algebra is improved if the operators have a simple and intuitive meaning. One aspect of this is that algebraic properties such as associativity should comply with the intuition of the operators.

- Efficient implementation: The detection mechanism should have a low overhead in terms of memory and execution time. For embedded and real-time systems it is vital that safe estimates of worst case memory usage and execution time can be derived statically.

1.1 Problem Formulation

The desired properties of an event algebra that are described above are all relatively straightforward to achieve in isolation. Many existing approaches, in particular those based on temporal logic or similar formalisms, are highly expressive and provide operators with intuitive properties, but in general this means that efficient implementation of event detection can not be achieved. Similarly, several event algebras are defined in terms of finite state machines which trivially ensures limited resource requirements, but typically at the cost of complicated and non-intuitive semantics for some operator combinations.

This thesis addresses the task of developing a formally defined event algebra for reactive systems that (i) complies with algebraic laws that intuitively ought to hold for the algebra operators, and (ii) permits an efficient implementation with limited resource requirements.

The problem statement is motivated by resource-conscious applications such as real-time and embedded systems. This type of systems

require that bounds for memory usage and execution time can be statically determined. Furthermore, they often appear in safety-critical applications for which formal verification is required. Providing laws that the algebra conforms to allows reasoning on a high level of abstraction, and facilitates verification.

1.2 The Approach

The operators of the proposed algebra, or variants of them, are basic operators found in many of the existing event algebras from different application domains. We believe that this choice of operators provides a good starting point. As future work, we plan to perform a thorough investigation of the expressiveness demands of the intended application domain to determine if the algebra would benefit from additional operators. This is discussed further in Section 8.1.

The algebra is defined by a set-based declarative semantics, rather than in terms of state automata, Petri nets or similar constructs. 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 issues in detail, and establish a simple relation between this algorithm and the declarative algebra semantics.

We use techniques such as interval-based semantics to preserve intuitive operator properties under operator composition, and a carefully designed restriction policy to deal with the memory complexity caused by some of the properties. These techniques are described further in Chapter 2.

The two objectives, intuitive algebraic properties and a bounded memory implementation, are contradictory to some extent. In situations where a trade-off has been unavoidable, the choice has been to prioritise algebraic properties over bounded memory. Consequently, rather than ensuring limited resource requirements in general, we have settled for identifying a class of expressions that can be correctly detected with limited resources. The aim when designing the algebra has then been to make this class as large as possible.

1.3 Related Publications

The event algebra presented in this thesis has evolved into the current form through a number of versions, some of which have been published.

- J. Carlson and B. Lisper, *An interval-based algebra for restricted event detection*. In Proceedings of the First International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS 2003), Marseille, France, September 2003.

  A first version of the algebra is presented in this paper. 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 presented, and the algebraic properties are weak compared to the current version, especially the relation between the unrestricted semantics and the result when the restriction policy is applied.

- J. Carlson and B. Lisper, *An improved algebra for restricted event detection*. MRTC Technical Report MDH-MRTC-159/2004-1-SE, February 2004.

  This paper improves the algebra by introducing temporal restriction, but for sequences only. For expressions where every sequence has a finite temporal restriction, limited memory requirement is ensured.

- J. Carlson and B. Lisper, *An event detection algebra for reactive systems*. Submitted, April 2004.

  The paper presents the same algebra version as in this thesis, but without the results on how information about the minimum separation time between primitive events can be used to achieve tighter resource bounds.

- J. Carlson and B. Lisper, *An event detection algebra for reactive systems*. MRTC Technical Report MDH-MRTC-117/2004-1-SE, April 2004.

  This technical report extends the previous paper with formal proofs.

1.4 Contributions

The main contributions of this thesis are:

- A novel event detection algebra that conforms to many algebraic laws that intuitively ought to hold for the algebra operators. These laws facilitate formal as well as informal reasoning about the algebra and the behaviour of a reactive system that uses it.

- A formal restriction policy that is used to establish the relation between the intuitive but inefficient algebra semantics, and a valid implementation thereof. The restriction policy is carefully designed to allow an efficient implementation while retaining the algebraic properties of the algebra semantics.

- An event detection algorithm that conforms to the algebra semantics with restriction applied, for which a large class of events can be detected with limited resources. In a time triggered setting, the algorithm provides a straightforward implementation of the algebra.

- A semantic preserving transformation algorithm, based on the algebraic laws for temporal restriction, that allows many expressions to be transformed to meet the criteria under which detection can be performed with limited resources.

- A prototype implementation in Java that provides an opportunity to test the algebra in practice, and illustrates some concerns related to implementing the algebra in an event triggered setting.

1.5 Organisation

The thesis is organised in the following way. Chapter 2 gives a brief introduction to concepts and techniques common to many event detection frameworks. The algebra in presented in Chapter 3. First, the declarative semantics of the algebra, including the restriction policy, is defined. Then, a number of important properties are proved, in particular the effect of applying the restriction policy in a nested fashion. This is followed by a description of an imperative detection algorithm in Chapter 4, together with a correctness result that establishes the relation between

the algorithm and the declarative semantics of the algebra. The chapter also contains an analysis of the memory and time complexity of the detection algorithm, and suggestions on how they can be improved.

Chapter 5 presents an event expression transformation algorithm for decreasing the memory needed to detect an event correctly, possibly from infinite to limited memory. We also show that the meaning of the event expression is preserved when the transformation algorithm is applied. In Chapter 6, a prototype implementation is presented in which the algebra is incorporated with the Java event system. Chapter 7 surveys related work, and a discussion followed by a description of future work in Chapter 8 concludes the thesis.

Chapter 2

Event Detection

Conceptually, the task of an event detection mechanism is to compute the occurrences of a given composite event from the occurrences of primitive events. The way in which the composite event is specified, and what is meant by an occurrence, differ between methods, as well as the type of event patterns that can be specified.

In some applications the event detection is performed on a finite collection of primitive event occurrences that was gathered in an earlier phase, for example as the result of monitoring a system or an environment. This allows the detection mechanism to process the data in arbitrary order and possibly in several passes, and typically do not impose hard resource constraints. Contrasting these off-line methods, reactive applications require events to be detected continually during the entire system lifetime (which might be infinite in theory). This implies that the detection mechanism has no knowledge of future occurrences of primitive events, and typically only limited information about past events can be stored due to resource restrictions.

Naturally, the term *event* means a different thing in different contexts. In particular, it is sometimes used to denote a single occurrence, and sometimes for one source or type of occurrences. In this thesis we distinguish between the two by referring to the former as an occurrence or *instance*. The latter is called an *event type*, or just event. Following this, the proposed algebra is an *event type algebra* rather than an *event instance algebra*, since the operators of the algebra combine simple event types into more complex event types.

When event detection is done by means of an event algebra, composite events are defined by expressions built recursively from primitive events and the operators of the algebra. The choice of operators differ between algebras, and is influenced by the type of systems for which the algebra is intended. Table 2.1 lists the operators used in this thesis, together with an informal description of their meaning. For formal definitions, see Section 3.1.

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

Table 2.1: Informal description of the algebra operators.

Some basic operators, such as disjunction, conjunction, sequence and negation, are found in many algebras, although their meaning might be slightly different. For example, the sequence operator might or might not allow partly overlapping events to be identified as a sequence, and conjunction is sometimes restricted to simultaneous occurrences. In addition to these common operators, the proposed algebra contains a temporal restriction construct that limits the length of the event occurrence. In algebras where this type of events can be specified, it is typically provided by variants of the ordinary operators, such as a temporally restricted sequence. Using an interval based semantics, described below, for our algebra allows a more general temporal restriction construct that can be applied to any event expression. The negation operator is also more general than what is provided by most algebras, as a result of the interval based semantics.

Example 2.1. The meaning of the negation operator and the temporal restriction construct is best understood by examples. The expression

$(A;B)-C$ denotes a composite event that occurs when an occurrence of A is followed by an occurrence of B an there is no occurrence of C in between. An event defined by the expression $(A;B)_\tau$ occurs when an occurrence of A is followed by an occurrence of B within τ time units. ◇

Example 2.2. The composite event E from Example 1.1 corresponds to the expression $(\mathsf{B};\mathsf{B})_2-(\mathsf{P}\vee\mathsf{T})$. ◇

2.1 Single Point or Interval Semantics

In most event algebras, each event occurrence, including events that require more than one occurrence of simpler events, is associated with a single time point (the time of detection, i.e., the time of the last occurrence that was required). Galton and Augusto [17] showed that this results in unintended semantics for some operator combinations, for example nested sequence operators, as described in Example 2.3. Inspired by methods in knowledge representation, they suggest that the problem can be solved by associating the occurrence of a complex event with the occurrence interval rather than the time of detection.

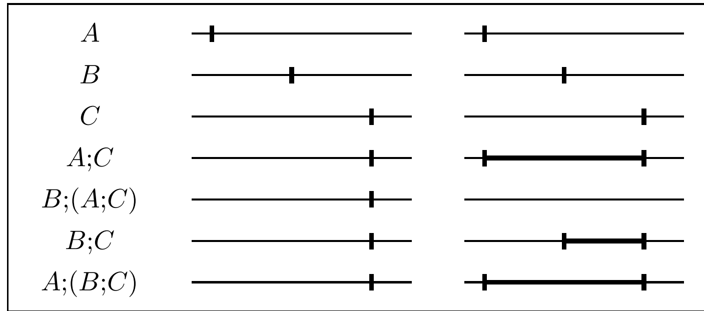


Figure 2.1: Comparison between single point semantics (left) and interval semantics (right).

Example 2.3. Figure 2.1 illustrate the difference between single point and interval semantics. In these figures, time flows from left to right, and

each row shows the occurrences of a primitive event type or the detected occurrences of an expression.

When single point detection is used, an instance of the event B;(A;C) is detected if A occurs first, and then B followed by C. The reason is that these occurrences cause a detection of A;C which is associated with the occurrence time of C. Since B occurs before this time point, an occurrence of B;(A;C) is detected. Figure 2.1 shows this situation in the left column, together with the intuitively correct detection of A;(B;C).

With interval semantics, 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 2.1. ◇

We base the event algebra on interval semantics, since it facilitates the design of operators that are intuitive also under composition. Instead of defining the occurrence interval for each operator, explicitly included in the operator semantics, we define the interval of an occurrence to be the smallest interval containing all primitive occurrences that caused it to occur.

2.2 Event Contexts

The operator semantics described informally above does not specify how to handle situations where an occurrence could participate in several occurrences of a composite event. For example, three occurrences of A followed by two occurrences of B result in six occurrences of $A+B$. While this may be acceptable, or even desirable, in some applications, the memory requirements (each occurrence of A and B must be remembered forever) and the increasing number of simultaneous events means that it is unsuitable in many cases.

A common way to modify the operator semantics to take this into account is by means of *event contexts*. First, each operator is given a simple meaning that defines the constraints on the participating occurrences that characterise the operator, similar to that of Table 2.1. Then a number of event contexts are defined that act as modifiers to the simple operator semantics. These contexts specify constraints on how occurrences may be selected when looking for occurrence patterns

that match the operator semantics. As a result, each combination of an operator and a context can be seen as a separate operator with a specific meaning.

Example 2.4. To illustrate the concept of event contexts as they are typically used, we define informally three of the contexts in Snoop [13], called unrestricted, recent and chronicle. To avoid details, we describe their effect on the sequence operator, rather than the general form that can be applied to any operator. When detecting $A;B$, the event contexts have the following meanings.

- Unrestricted: All instances of A and B are valid.
- Recent: If an instance of B can be combined with several instances of A to form instances of $A;B$, only the most recent instance of A is valid.
- Chronicle: If an instance of B can be combined with several instances of A to form instances of $A;B$, only the oldest instance of A is valid. Also, this instance is never valid in the future.

Figure 2.2 shows the effect of these contexts on the sequence operator. ◇

In many existing event algebras the event contexts are only defined informally. Also, carelessly defined contexts might work as intended for some operators but introduce unintended effects for other [40].

Event contexts provide variants of the algebra operators, to be used by a developer of a reactive system to achieve a more specific behaviour than what is specified by the operator semantics. Some contexts also affect the resource requirements of the operator to which it is applied. For example, in the recent context only the most recent instance of each constituent type must be stored for future use, and thus all basic operators can be implemented with limited resources in this context. Unfortunately, event contexts typically ruin many of the algebraic properties that hold for the simple operator semantics.

The restriction policy proposed in this thesis was originally influenced by this type of event contexts, but the conceptual role of the restriction policy is different. We consider the intuitive and simple operator semantics to be the intended behaviour of the event detection, but due to

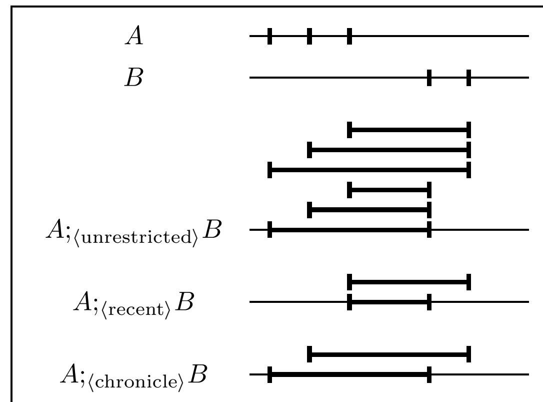


Figure 2.2: Comparison of three event contexts (unrestricted, recent and chronicle) for the sequence operator.

efficiency considerations only a subset of these occurrences can be detected. We use a restriction policy to formalise how this subset may be selected. Thus, the restriction policy is conceptually applied once to the event expression as a whole, and not to the individual operators.

Chapter 3

The Event Algebra

As described in the introduction, the algebra is defined by a declarative semantics based on sets. We also introduce a formal restriction policy that defines what is considered a valid implementation of the algebra. Once the algebra is defined, we investigate the algebraic properties of the operators and the restriction policy.

For simplicity, we assume a discrete time model throughout the thesis. The declarative semantics of the algebra can 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. We also assume that occurrences of primitive events are instantaneous, and that each primitive event occurs at most once each time instant.

3.1 Declarative Semantics

Before defining the syntax and semantics of the algebra we define concepts needed to represent primitive events and their occurrences. These concepts are then extended to encompass composite events as well.

3.1.1 Primitive Events

We assume that the system has a pre-defined set of primitive event types to which it should be able to react. These events can be external (sampled from the environment or originating from another system) or internal (such as the violation of a condition over the system state, or

a timeout), but the detection mechanism does not distinguish between these categories.

For some primitive events, it is useful to associate additional information with each occurrence. For example, the occurrences of a temperature alarm might carry the measured temperature value, to be used in the responding action. These values are not manipulated by the algebra, only grouped and forwarded to the part of the system that reacts to the detected events.

Definition 3.1. Let $\mathcal{P}$ be a finite set of identifiers that represent the primitive event types that are available to the system. For each identifier $p \in \mathcal{P}$, let $\text{dom}(p)$ denote the value domain of p, i.e., the values that can be associated with instances of p.

Definition 3.2. The temporal domain $\mathcal{T}$ is the set of natural numbers.

Occurrences of primitive events are assumed to be instantaneous and atomic. In the algebra, they are represented by event instances that contain event type, a value and occurrence time. Formally, we represent a primitive instance as a singleton set, to allow primitive and complex instances to be treated uniformly.

Definition 3.3. If $p \in \mathcal{P}$, $v \in \text{dom}(p)$ and $\tau \in \mathcal{T}$, then the singleton set $\{\langle p, v, \tau \rangle\}$ is a primitive event instance.

Together, the occurrences of a certain event type form an event stream.

Definition 3.4. A primitive event stream is a set of primitive event instances all of which have the same identifier and different times.

Both the set of identifiers and the value domains capture static aspects of the system. Instances and event streams, however, are dynamic concepts that describe what happens during a particular scenario. An interpretation is a formal representation of a single scenario, as it describes one of the possible ways in which the primitive event can occur.

Definition 3.5. An interpretation is a function that maps each identifier $p \in \mathcal{P}$ to a primitive event stream containing instances with identifier p.

Example 3.1. For the system in the previous examples, we assume that instances of the temperature alarm T carry temperature measurements

represented by natural numbers. The pressure alarm P is less sensitive and these instances only contain information about whether the pressure is too low or too high. The button B instances do not carry any additional information, which is represented by a dummy element $\perp$. These static aspects of the system can be captured formally by $\mathcal{P} = \{\mathsf{T}, \mathsf{P}, \mathsf{B}\}$, with $\mathrm{dom}(\mathsf{T}) = \mathbb{N}$, $\mathrm{dom}(\mathsf{P}) = \{\mathrm{high}, \mathrm{low}\}$ and $\mathrm{dom}(\mathsf{B}) = \{\perp\}$.

As an example of a particular scenario, we consider an interpretation $\mathcal{I}$ such that $\mathcal{I}(\mathsf{T}) = S$, $\mathcal{I}(\mathsf{P}) = S'$ and $\mathcal{I}(\mathsf{B}) = \emptyset$, where S and S' are the following primitive event streams:

$$S = \{\{\langle \mathsf{T}, 12, 2\rangle\},\ \{\langle \mathsf{T}, 14, 3\rangle\},\ \{\langle \mathsf{T}, 8, 5\rangle\}\} \quad \text{and} \quad S' = \{\{\langle \mathsf{P}, \mathrm{low}, 4\rangle\}\}$$

◇

3.1.2 Composite Events

Composite events are represented by expressions built recursively from the identifiers and the operators of the algebra.

Definition 3.6. If $A \in \mathcal{P}$, then A is an *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 event expressions.

Next, we extend the concepts of instances and streams to composite events. An instance of a composite event is always triggered by one or more instances of simpler events, and the information associated with these simpler instances should somehow be included in the representation of the composite event instance.

One design decision is whether the structure of the expression should be visible in the representation of its instances, or not. For simplicity, we use a flat instance representation that is independent from the structure of the expression. Informally, an instance of a composite event will consist of all the primitive event occurrences that caused it to occur, either directly or indirectly by causing simpler composite events to occur. Also, there is no explicit information in an instance about which event type it is an instance of. This is implicitly provided by the events stream to which the instance belongs.

As an example, consider an instance of $A;B$ that is caused by an instance a of A and one instance b of B. This instance will be represented

by the set $a \cup b$. An alternative where the expression structure is visible in the instances would be to represent this instance by $\langle a, b \rangle$.

The way in which instances are constructed is defined by the algebra semantics. For now, we only define their structure.

Definition 3.7. An event instance is a non-empty union of primitive event instances.

Since the semantics should be interval-based, we associate each instance with an interval, through the following definition.

Definition 3.8. For an event instance a we define

$$\begin{aligned} \text{start}(a) &= \min(\,\{\tau \mid \langle p, \upsilon, \tau \rangle \in a\}\,) \\ \text{end}(a) &= \max(\,\{\tau \mid \langle p, \upsilon, \tau \rangle \in a\}\,) \end{aligned}$$

The interval $[\text{start}(a), \text{end}(a)]$ can be thought of as the smallest interval which contains all the occurrences of primitive events that caused a to occur. Note that a primitive event instance is an event instance, and if a is a primitive event instance then $\text{start}(a) = \text{end}(a)$.

Example 3.2. Let $a = \{\langle \mathsf{T}, 12, 2\rangle,\ \langle \mathsf{P}, \text{low}, 4\rangle,\ \langle \mathsf{T}, 8, 5\rangle\}$. Then a is an event instance, and we have $\text{start}(a) = 2$ and $\text{end}(a) = 5$. ◇

In the graphical notation used in the examples, composite event instances are visualised by start and end time only. In cases where more details are required, the times of all primitive instances in the composite event instance are marked.

We also need a definition of general event streams. These will be used to represent all instances of a composite event. By this definition, a primitive event stream is an event stream, just as the names suggest.

Definition 3.9. An event stream is a set of event instances.

The variable naming convention used in the thesis is to use S, T and U for event streams, and A, B, C, etc. for event expressions. Lower case letters are used for event instances, and in general s belongs to the event stream S, etc.

3.1.3 Semantics

The interpretation provides the occurrences of each primitive event, by mapping each identifier to an event stream. The role of the algebra semantics is to extend this mapping to composite events defined by event expressions. The following functions on event streams form the core of the algebra semantics, defining the characteristics of the five operators.

Definition 3.10. For event streams S and T, and $\tau \in \mathcal{T}$, we define:

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

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

Definition 3.11. The meaning of an event expression for a given interpretation $\mathcal{I}$ is defined as follows:

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

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 3.3. Consider the expression $\mathsf{T};\mathsf{P}$. According to the algebra semantics, the meaning of this expression is

$$[\![\mathsf{T};\mathsf{P}]\!]^{\mathcal{I}} = \mathrm{seq}([\![\mathsf{T}]\!]^{\mathcal{I}}, [\![\mathsf{P}]\!]^{\mathcal{I}}) = \mathrm{seq}(\mathcal{I}(\mathsf{T}), \mathcal{I}(\mathsf{P}))$$

For the scenario captured by the interpretation in Example 3.1, the concrete meaning of the expressions is

$$[\![\mathsf{T};\mathsf{P}]\!]^{\mathcal{I}} = \{\{\langle \mathsf{T}, 12, 2\rangle, \langle \mathsf{P}, \mathrm{low}, 4\rangle\},\ \{\langle \mathsf{T}, 14, 3\rangle, \langle \mathsf{P}, \mathrm{low}, 4\rangle\}\}$$

Figure 3.1 illustrates this graphically. $\diamond$

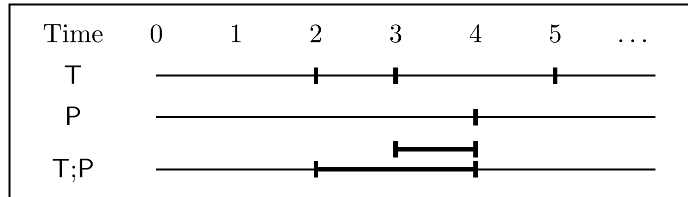


Figure 3.1: Graphical representation of Example 3.3.

The algebra semantics is reasonably intuitive and simple enough to aid formal as well as informal reasoning about the meaning of expressions. The operators behave properly also in complex, nested expressions, which is captured by the algebraic laws presented in Section 3.2. However, the algebra can not be efficiently implemented in this form, as there are no bounds on the number of simultaneous instances, nor on the memory required to store instances for future use.

To deal with this, we expect an implementation to detect only a subset of the instances specified by the algebra semantics given above. Naturally, allowing implementations to detect any subset is not very constructive. Instead, we introduce a formal restriction policy that defines what is considered a valid subset for an implementation to detect. Conceptually, this restriction policy is applied to the expression as a whole, but it is designed to ensure that this semantically consistent with applying it recursively to all subexpressions, which is required to allow an efficient implementation.

Ideally, the restriction policy should interfere as little as possible with the properties of the unrestricted semantics. None of the removed instances should have a crucial impact on the detection of enclosing expressions. At the same time, operators such as conjunction and sequence must be able to identify non-valid instances early, before the end time of the instance is reached, in order not to waste memory.

Our restriction policy is defined as a predicate and not as a function. Alternatively, it can be seen as a non-deterministic restriction function, or a family of valid restriction functions. For reasons of repeatability, it is desirable that an implementation of the algebra is deterministic. From a theoretical point of view, however, we prefer to leave open as many detailed design decisions as possible, since we can still ensure that any implementation which is consistent with the restriction policy predicate

is guaranteed to have the properties described in this thesis. This design decision is motivated by the increased flexibility it provides when implementing the algebra. Choices that are non-deterministic in the formal definition can be made on the basis of implementation details to increase efficiency.

The basis of the restriction policy it that the restricted event stream should be a subset that does not contain multiple instances with the same end time. Informally, from the instances with the same end time, the restriction policy keeps exactly one with maximal start time. Formally, the restriction policy is defined as follows.

Definition 3.12. For two event streams, S and S', $\mathit{rem}(S, S')$ 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. $\forall s, s'((s \in S' \wedge s' \in S' \wedge \text{end}(s) = \text{end}(s')) \Rightarrow s = s')$

In Section 3.2.3 we show that this restriction policy can be applied recursively to all subexpressions of an event expression with a well defined impact on the resulting event stream. Section 4.2 argues that the restricted version of the algebra can be efficiently implemented.

Example 3.4. Figure 3.2 illustrates the result of applying the restriction policy to an event stream S. From the three instances of S with end time 4 the one with start time 1 must be removed, together with one of the two with start time 2. For the two instances that end at time 7, the one with earliest start time must be removed. The long instance is the only instance ending at time 8, and thus it must be included in the restricted stream.

The choice of which of the two short instances with end time 4 to remove results in two valid restrictions of the event stream S, named S' and S'' in the figure. It is straightforward to see that $\mathit{rem}(S, S')$ and $\mathit{rem}(S, S'')$ holds, and that there is no other event stream T such that $\mathit{rem}(S, T)$ holds. $\diamond$

Event contexts, for example those presented in Section 2.2, are typically defined in terms of conditions on the constituent event instances. The restriction policy defined in this thesis differs from these in that it

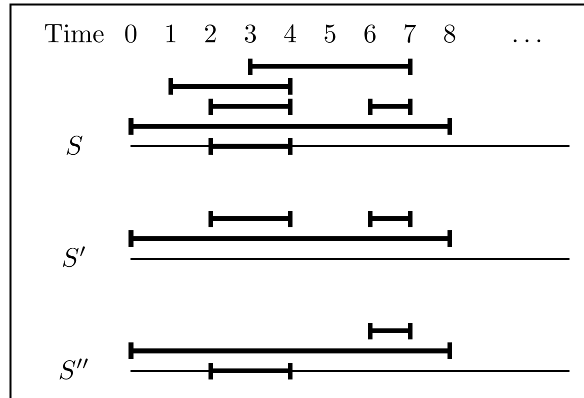


Figure 3.2: An event stream S with two valid restrictions, i.e., both $rem(S, S')$ and $rem(S, S'')$ holds.

is explicitly applied to the event stream produced by the unrestricted algebra semantics. This results in a simpler restriction policy semantics, at the cost of reduced expressiveness when designing the policy, since restriction decisions must be based solely on the information in the event stream. For example, we would not be able to modify the policy to give priority to the left argument of a disjunction, unless the instance representation is changed to include additional information.

For the sake of completeness, we show that this restriction policy is constructive, i.e., that for any event stream there exists a valid restriction.

Theorem 3.1.1. For any event stream S, there exists an event stream S' such that $rem(S, S')$.

Proof. The discrete time model ensures that there is at least one instance with maximal start time in any subset of S. Thus, there is always at least one way to select which of the instances with the same end time to include in the restricted stream. □

In a dense time setting, S could contain an infinite sequence of increasingly shorter instances with the same end time. Then, there is no instance with maximal start time to include in the restricted stream. If the definition of primitive event stream is limited by the additional

condition that for any finite time interval there is only a finite number of instances with times within that interval, then the theorem holds for a dense time model as well.

3.2 Properties

We have argued that the algebra semantics defined in the previous section corresponds to the intuitive meaning of the operators, but intuition is personal and in many cases inconsistent, and other considerations sometimes conflict with what is intuitively valid. To aid a user of the algebra, this section presents a number of useful laws that the algebra complies with. These laws facilitate formal and informal reasoning about the algebra and the system in which it is embedded, and show to what extent the operators behave according to intuition.

We also investigate how these laws are affected by the restriction policy, and the result of applying restriction recursively to all subexpressions of an expression. The latter is crucial for implementing the algebra with limited resources. First, however, a notion of expression equivalence is defined.

Definition 3.13. Two event expressions A and B are equivalent (denoted $A \equiv B$) iff $[\![A]\!]^{\mathcal{I}} = [\![B]\!]^{\mathcal{I}}$ for any interpretation $\mathcal{I}$.

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

Theorem 3.2.1. 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 in a straightforward way from Definitions 3.10 and 3.13. □

3.2.1 Algebraic Laws

The following laws describe the properties of the disjunction, conjunction and sequence operators, and how they distribute.

Theorem 3.2.2. For event expressions A, B and C, the following laws

hold.

$$
\begin{array}{lrcl}
1. & A\vee A & \equiv & A \\
2. & A\vee B & \equiv & B\vee A \\
3. & A+B & \equiv & B+A \\
4. & A\vee(B\vee C) & \equiv & (A\vee B)\vee C \\
5. & A+(B+C) & \equiv & (A+B)+C \\
6. & A;(B;C) & \equiv & (A;B);C \\
7. & (A\vee B)+C & \equiv & (A+C)\vee(B+C) \\
8. & (A\vee B);C & \equiv & (A;C)\vee(B;C) \\
9. & A;(B\vee C) & \equiv & (A;B)\vee(A;C)
\end{array}
$$

Corollary 3.2.1.

$$
10. \quad A+(B\vee C) \equiv (A+B)\vee(A+C)
$$

Proof. Most of the laws follow in a straightforward way from Definitions 3.13, 3.10 and 3.11.

1. $\llbracket A\vee A\rrbracket = \mathrm{dis}(\llbracket A\rrbracket, \llbracket A\rrbracket) = \llbracket A\rrbracket \cup \llbracket A\rrbracket = \llbracket A\rrbracket$

2. $\llbracket A\vee B\rrbracket = \mathrm{dis}(\llbracket A\rrbracket, \llbracket B\rrbracket) = \mathrm{dis}(\llbracket B\rrbracket, \llbracket A\rrbracket) = \llbracket B\vee A\rrbracket$

3. $\llbracket A+B\rrbracket = \mathrm{con}(\llbracket A\rrbracket, \llbracket B\rrbracket) = \mathrm{con}(\llbracket B\rrbracket, \llbracket A\rrbracket) = \llbracket B+A\rrbracket$

4. $\llbracket A\vee(B\vee C)\rrbracket = \llbracket A\rrbracket \cup \llbracket B\rrbracket \cup \llbracket C\rrbracket = \llbracket (A\vee B)\vee C\rrbracket$

5. $\llbracket A+(B+C)\rrbracket = \mathrm{con}(\llbracket A\rrbracket, \mathrm{con}(\llbracket B\rrbracket, \llbracket C\rrbracket)) =$
   $\{a \cup b \cup c \mid a \in \llbracket A\rrbracket \wedge b \in \llbracket B\rrbracket \wedge c \in \llbracket C\rrbracket) = \llbracket (A+B)+C\rrbracket$

6. $\llbracket A;(B;C)\rrbracket = \{a \cup e \mid a\in\llbracket A\rrbracket \wedge e\in\{b \cup c \mid b\in\llbracket B\rrbracket \wedge c\in\llbracket C\rrbracket \wedge$
   $\mathrm{end}(b)<\mathrm{start}(c)\} \wedge \mathrm{end}(a)<\mathrm{start}(e)\} = \{a \cup b \cup c \mid a\in\llbracket A\rrbracket \wedge$
   $b\in\llbracket B\rrbracket \wedge c\in\llbracket C\rrbracket \wedge \mathrm{end}(a)<\mathrm{start}(b) \wedge \mathrm{end}(b)<\mathrm{start}(c)\} = \llbracket (A;B);C\rrbracket$

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

8. $\llbracket (A\vee B);C\rrbracket = \{e \cup c \mid e\in\llbracket A\rrbracket \cup \llbracket B\rrbracket \wedge c\in\llbracket C\rrbracket \wedge \mathrm{end}(e)<\mathrm{start}(c)\} =$
   $\{a \cup c \mid a\in\llbracket A\rrbracket \wedge c\in\llbracket C\rrbracket \wedge \mathrm{end}(a)<\mathrm{start}(c)\} \cup$
   $\{b \cup c \mid b\in\llbracket B\rrbracket \wedge c\in\llbracket C\rrbracket \wedge \mathrm{end}(b)<\mathrm{start}(c)\} = \llbracket (A;C)\vee(B;C)\rrbracket$

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

10. This follows from laws 2, 3 and 7.

□

Next, we present a set of laws for negation. To simplify the proofs, we introduce the following predicate.

Definition 3.14. For an event stream S, and time instants $\tau, \tau' \in \mathcal{T}$, define *empty*$(S,\tau,\tau')$ to hold if $\neg\exists s(s\in S \wedge \tau\leq\text{start}(s) \wedge \text{end}(s)\leq\tau')$.

Proposition 3.2.1.

i. $a\in[\![A-B]\!]$ iff $a\in[\![A]\!]$ and *empty*$([\![B]\!],\text{start}(a),\text{end}(a))$.

ii. *empty*$(S\cup S',\tau,\tau')$ iff *empty*(S,τ,τ') and *empty*(S',τ,τ')

iii. If $\tau_1 \leq \tau_1'$ and $\tau_2' \leq \tau_2$, then *empty*(S,τ_1,τ_2) implies *empty*(S,τ_1',τ_2')

Proof. The properties follow trivially from the definition. □

Theorem 3.2.3. For event expressions A, B and C, the following laws hold.

11.	$(A-B)-C$	$\equiv$	$A-(B\vee C)$
12.	$(A\vee B)-C$	$\equiv$	$(A-C)\vee(B-C)$
13.	$(A+B)-C$	$\equiv$	$((A-C)+B)-C$
14.	$(A;B)-C$	$\equiv$	$((A-C);B)-C$
15.	$(A;B)-C$	$\equiv$	$(A;(B-C))-C$

Corollary 3.2.2.

16.	$(A-B)-B$	$\equiv$	$A-B$
17	$(A-B)-C$	$\equiv$	$(A-C)-B$
18.	$(A\vee B)-C$	$\equiv$	$((A-C)\vee B)-C$
19.	$(A\vee B)-C$	$\equiv$	$(A\vee(B-C))-C$
20.	$(A+B)-C$	$\equiv$	$(A+(B-C))-C$
21.	$(A-B)-C$	$\equiv$	$((A-C)-B)-C$

Proof. Here, $\equiv^{23}$ denotes that the equivalence follows from law number 23, etc. Similarly, $=^{\text{i}}$ or $\Leftrightarrow^{\text{ii}}$ denotes that the equivalence is based on the corresponding property in Proposition 3.2.1.

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

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

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

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

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

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

16. This follows from laws 1 and 12.
17. This follows from laws 2 and 11.
18. $((A{-}C){\vee}B){-}C \equiv^{12} ((A{-}C){-}C){\vee}(B{-}C) \equiv^{16} (A{-}C){\vee}(B{-}C) \equiv^{12} (A{\vee}B){-}C$
19. This follows from laws 2 and 12.
20. This follows from laws 3 and 13.
21. $((A{-}C){-}B){-}C \equiv^{17} ((A{-}B){-}C){-}C \equiv^{16} (A{-}B){-}C$

□

Next, we present laws describing how temporal restrictions can be propagated through an expression. These laws are used in Chapter 5 to construct an algorithm for transforming event expressions into equivalent expressions that can be detected more efficiently.

Theorem 3.2.4. For event expressions A, B and C, and $\tau \in \mathcal{T}$, the following laws hold.

$$
\begin{array}{rrcl}
22. & A & \equiv & A_\tau \quad \text{if } A \in \mathcal{P} \\
23. & (A_\tau)_{\tau'} & \equiv & A_{\min(\tau,\tau')} \\
24. & (A{\vee}B)_\tau & \equiv & A_\tau{\vee}B_\tau \\
25. & (A{+}B)_\tau & \equiv & (A_\tau{+}B)_\tau \\
26. & (A{-}B)_\tau & \equiv & (A_\tau){-}B \\
27. & (A{-}B)_\tau & \equiv & (A{-}B_\tau)_\tau \\
28. & (A;B)_\tau & \equiv & (A_\tau;B)_\tau \\
29. & (A;B)_\tau & \equiv & (A;B_\tau)_\tau
\end{array}
$$

Corollary 3.2.3.

$$
\begin{array}{rrcl}
30. & (A_\tau)_{\tau'} & \equiv & (A_{\tau'})_\tau \\
31. & (A{\vee}B)_\tau & \equiv & ((A_\tau) \vee B)_\tau \\
32. & (A{\vee}B)_\tau & \equiv & (A \vee (B_\tau))_\tau \\
33. & A_\tau{\vee}B_{\tau'} & \equiv & (A_\tau \vee B_{\tau'})_{\max(\tau,\tau')} \\
34. & (A{+}B)_\tau & \equiv & (A{+}B_\tau)_\tau \\
35. & (A{-}B)_\tau & \equiv & A_\tau{-}B_\tau
\end{array}
$$

Proof.

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

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

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

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

26. $[\![(A-B)_\tau]\!] = \{a \mid a\in[\![A-B]\!] \wedge \text{end}(a)-\text{start}(a) \leq \tau\} =$
$\{a \mid a\in[\![A]\!] \wedge \text{end}(a)-\text{start}(a) \leq \tau \wedge$
$\neg\exists b(b\in[\![B]\!] \wedge \text{start}(a)\leq\text{start}(b) \wedge \text{end}(b)\leq\text{end}(a))\} =$
$\{a \mid a\in[\![A_\tau]\!]\wedge\neg\exists b(b\in[\![B]\!]\wedge\text{start}(a)\leq\text{start}(b)\wedge\text{end}(b)\leq\text{end}(a))\} =$
$[\![A_\tau-B]\!]$

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

28. $[\![(A;B_\tau)_\tau]\!] =$
$\{a\cup b \mid a\in[\![A]\!]\wedge b\in[\![B_\tau]\!]\wedge\text{end}(a)<\text{start}(b)\wedge\text{end}(b)-\text{start}(a) \leq \tau\} =$

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

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

30. $(A_\tau)_{\tau'} \equiv^{27} A_{\min(\tau,\tau')} \equiv A_{\min(\tau,\tau')} \equiv^{27} (A_{\tau'})_\tau$

31. $(A \vee B)_\tau \equiv^{24} A_\tau \vee B_\tau \equiv^{23} (A_\tau)_\tau \vee B_\tau \equiv^{24} (A_\tau \vee B)_\tau$

32. This follows from laws 2 and 31.

33. $(A_\tau \vee B_{\tau'})_{\max(\tau,\tau')} \equiv^{24} (A_\tau)_{\max(\tau,\tau')} \vee (B_{\tau'})_{\max(\tau,\tau')} \equiv^{23} A_{\min(\tau,\max(\tau,\tau'))} \vee B_{\min(\tau',\max(\tau,\tau'))} \equiv A_\tau \vee B_{\tau'}$

34. This follows from laws 3 and 25.

35. $(A - B)_\tau \equiv^{27} (A - B_\tau)_\tau \equiv^{26} A_\tau - B_\tau$

□

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

Definition 3.15. Let 0 denote the empty event, semantically defined as $[\![0]\!]^{\mathcal{I}} = \emptyset$ for any interpretation $\mathcal{I}$.

Theorem 3.2.5. For an event expression A and $\tau \in \mathcal{T}$, the following laws hold.

$$
\begin{array}{lrcl}
36. & 0 \vee A & \equiv & A \\
37. & 0 + A & \equiv & 0 \\
38. & A - A & \equiv & 0 \\
39. & 0 - A & \equiv & 0 \\
40. & A - 0 & \equiv & A \\
41. & 0; A & \equiv & 0 \\
42. & A; 0 & \equiv & 0 \\
43. & 0_\tau & \equiv & 0
\end{array}
$$

Proof. These laws follow in a straightforward way from the definition of 0 and the operator semantics. □

3.2.2 Impact from the Restriction Policy on the Laws

The laws consider equivalence between expressions with respect to the algebra semantics. However, in an implementation where the restriction policy is applied, equivalent expressions might produce different results since the non-deterministic choices in the restriction policy might depend on the structure of the expression in an implementation.

Example 3.5. Consider the event stream S from Example 3.4, and imagine two equivalent event expressions $A \equiv A'$ with $[\![A]\!] = [\![A']\!] = S$. Since S' and S'' are both valid restrictions of S, it might be that an implementation of the algebra results in S' when detecting A, and in S'' when detecting A'. ◇

Consequently, it should be clarified to what extent the laws presented above are still applicable when restriction is applied.

Theorem 3.2.6. If $A \equiv A'$ and $rem([\![A]\!], S)$ holds, then $rem([\![A']\!], S)$ holds as well.

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

Thus, $A \equiv A'$ ensures that the result of an implementation detecting A is always a valid result for A'. As long as reasoning is based on the algebra semantics and the restriction policy, and not on the details of a particular detection algorithm such as the one presented in Section 4.1, it will be equally valid for equivalent expressions.

Example 3.6. In the previous example, according to the algebra semantic and the restriction policy, S'' is a perfectly valid result for A'. Reasoning about the system should not be based on the fact that the implementation happened to result in S' when detecting A. $\diamond$

To further investigate the relation between equivalent expressions when restriction is applied, notice that the restriction policy implies that detected event streams for equivalent expressions always contain instances with corresponding start and end times. This means that the part of the system that responds to the detected event occurrences is notified at the same time for equivalent expressions, but possibly with different values attached to the detected occurrences. Formally, we express this as follows.

Definition 3.16. For event streams S and T, define $S \cong T$ to hold if $\{\langle \text{start}(s), \text{end}(s)\rangle \mid s \in S\} = \{\langle \text{start}(t), \text{end}(t)\rangle \mid t \in T\}$

Trivially, $\cong$ is an equivalence relation.

Theorem 3.2.7. If $rem(S,T)$ and $rem(S,T')$ holds, then $T \cong T'$

Proof. Take any $t \in T$. Then, since $T \subseteq S$, $t \in S$. By the second condition in the definition of *rem*, 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 *rem* this implies $t = t''$, which means that we have $\text{start}(t) \leq \text{start}(t') \leq \text{start}(t)$ and thus $\text{start}(t') = \text{start}(t)$. So, for any $t \in T$ there is a $t' \in T'$ with the same start and end time. Trivially, the opposite holds as well. $\square$

Corollary 3.2.4. If $A \equiv A'$, $rem([\![A]\!],T)$ and $rem([\![A']\!],T')$ holds, then $T \cong T'$.

Proof. This follows from the theorem since $A \equiv A'$ by definition implies $[\![A]\!] = [\![A']\!]$. $\square$

Thus, $A \equiv A'$ ensures that for any implementation consistent with the restriction policy, the instances found when detecting A and A' have the same start and end times. This means that the part of the system that responds to the detected event occurrences is notified at the same time for equivalent expressions, but possibly with different values attached to the detected occurrences.

3.2.3 Properties of the Restriction Policy

In order to achieve the desired efficiency, all subexpressions of an expression must be detected in an efficient way. This requires that the restriction policy is applied not only to the whole expression but recursively to every subexpression, resulting in a far more complicated semantics than the one presented so far.

In general, this would 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 operators and the restriction policy have been carefully designed to support the following theorem. Informally, it states that introducing restriction of the subexpressions gives a result which is valid also for the case when restriction is applied only at the top level. The opposite does not hold, however. The set of valid restricted streams when restriction is applied recursively is a subset of the streams that are valid for single top-level restriction. This is illustrated by Example 3.7 below. The theorem is used in Section 4.1 to prove the correctness of the detection algorithm.

Theorem 3.2.8. If $rem(S, S')$ and $rem(T, T')$ holds, then for any event stream U and $\tau \in \mathcal{T}$ the following implications hold:

$$\begin{array}{lllll} \text{i.} & rem(\text{dis}(S', T'),\ U) & \Rightarrow & rem(\text{dis}(S, T),\ U) \\ \text{ii.} & rem(\text{con}(S', T'),\ U) & \Rightarrow & rem(\text{con}(S, T),\ U) \\ \text{iii.} & rem(\text{neg}(S', T'),\ U) & \Rightarrow & rem(\text{neg}(S, T),\ U) \\ \text{iv.} & rem(\text{seq}(S', T'),\ U) & \Rightarrow & rem(\text{seq}(S, T),\ U) \\ \text{v.} & rem(\text{tim}(S', \tau),\ U) & \Rightarrow & rem(\text{tim}(S, \tau),\ U) \end{array}$$

Proof.

i. Assume $rem(\text{dis}(S', T'), U)$. For any $u \in U$ we have $u \in \text{dis}(S', T')$ and thus $u \in S' \cup T'$. Then, since $S' \subseteq S$ and $T' \subseteq T$, we have $u \in S \cup T$, implying $u \in \text{dis}(S, T)$. Thus $U \subseteq \text{dis}(S, T)$, which satisfies the first constraint in the definition of *rem*.

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

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

ii. Assume $rem(\text{con}(S',T'),U)$. For any $u \in U$ we have $u \in \text{con}(S',T')$ and thus $u = s \cup t$ with $s \in S'$ and $t \in T'$. By the subset requirement in the definition of *rem*, $s \in S$ 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 $u = s \cup t$ with $s \in S$ and $t \in T$, and by the definition of *rem* 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 $u' = s' \cup t'$. 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 *rem*.

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

iii. Assume $rem(\text{neg}(S',T'),U)$. 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 *rem*, $u \in S$. 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 $u \in \text{neg}(S,T)$. Then $u \in S$ and 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 *rem*.

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

iv. Assume $rem(\text{seq}(S',T'),U)$. For any $u \in U$ we have $u \in \text{seq}(S',T')$ and thus $u = s \cup t$ with $s \in S'$, $t \in T'$ and $\text{end}(s) < \text{start}(t)$. By the subset requirement in the definition of *rem*, $s \in S$ 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 $u = s \cup t$ such that $s \in S$, $t \in T$ and $\text{end}(s) < \text{start}(t)$. By the definition of *rem* there exists $s' \in S'$ and $t \cup T'$ with $\text{start}(s) \le \text{start}(s')$, $\text{end}(s') = \text{end}(s)$, $\text{start}(t) \le \text{start}(t')$ and $\text{end}(t') = \text{end}(t)$. Let $u' = s' \cup t'$. Now, since $\text{end}(s') = \text{end}(s) < \text{start}(t) \le \text{start}(t')$, we have $u' \in \text{seq}(S',T')$ and $\text{start}(u) \le \text{start}(u')$ and $\text{end}(u') = \text{end}(u)$. This means that there exists some $u'' \in U$ with $\text{start}(u) \le \text{start}(u'')$ and $\text{end}(u'') = \text{end}(u)$, which satisfies the second constraint in the definition of *rem*.

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

v. Assume $rem(\text{tim}(S',\tau),U)$. For any $u \in U$ we have $u \in \text{tim}(S',\tau)$ and thus $u \in S'$ and $\text{end}(u) - \text{start}(u) \le \tau$. By the subset requirement in the definition of *rem*, we have $u \in S$ which means that $u \in \text{tim}(S,\tau)$ and thus $U \subseteq \text{tim}(S,\tau)$.

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

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

□

Example 3.7. This example illustrate that the implications in Theorem 3.2.8 do not hold in the opposite direction. Consider the event stream $S = [\![\mathsf{P};(\mathsf{B};\mathsf{T})]\!]$, and an interpretation consisting of the four non-overlapping event instances p, b_1, b_2 and t occurring in this order, named after the identifier to which they belong. Figure 3.3 depicts this scenario.

Clearly, $S' = \{p \cup b_1 \cup t\}$ is a valid restriction of S, i.e., $rem(S,S')$. For the case of multiple restrictions, let $T = [\![\mathsf{B};\mathsf{T}]\!]$. No T' for which $rem(T,T')$ holds can contain $b_1 \cup t$. As a result, $\text{seq}([\![\mathsf{P}]\!],T')$ can not contain the instance $p \cup b_1 \cup t$. Thus, one of the streams that are valid when the restriction policy is applied once is not valid for recursive application of restriction. ◇

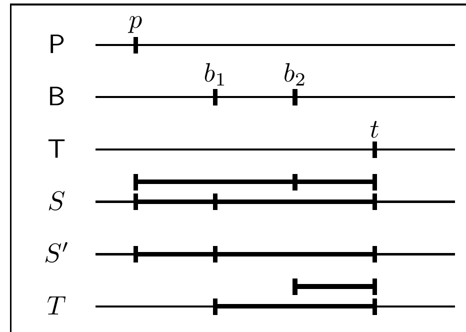


Figure 3.3: Graphical representation of Example 3.7.

The following example illustrates that the fact that restriction is based on start times is crucial to achieve good properties when restriction is applied recursively.

Example 3.8. Consider the event streams S and T depicted in Figure 3.4 together with the stream for the corresponding negation. For T, we have a single valid restriction T'. An important property of the policy is that replacing T in the negation by the restricted stream T', does not introduce additional instances. If we consider instead an imaginary restriction policy, for which T'' is a valid restriction to T, the resulting event stream contains instances not found in the unrestricted variant. ◇

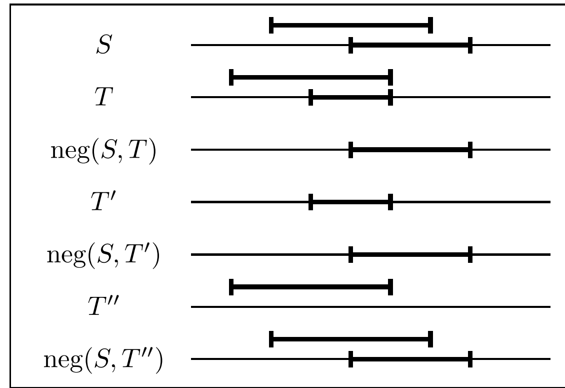


Figure 3.4: Graphical representation of Example 3.8.

Chapter 4

An Event Detection Algorithm

The simplicity of the declarative semantics is very helpful when investigating the properties of the algebra, as shown in the previous chapter. However, it does not provide much insight in whether the algebra can be effectively implemented, or how an implementation could be constructed. In this chapter, we present an imperative algorithm for detecting an event defined by a given event expression. This algorithm is proven correct with respect to the declarative algebra semantics and the restriction policy, and analysed for time and memory complexity.

Definition 4.1. Throughout this chapter, E denotes the event expression that is to be detected. The numbers $1 \dots m$ are assigned to the subexpressions of E in bottom-up order, and we let E^i denote subexpression number i. Consequently, $E^m = E$.

4.1 The Algorithm

Figure 4.1 presents the algorithm for detecting the event defined by the event expression E. The algorithm is executed once every time instant, and computes the current instance of E from the current instances of the primitive events, and from stored information about the past.

Variables are indexed from 1 to m since each operator in the expression requires its own state variables. The variable 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 l_i, r_i, t_i and q_i store information about the past needed to detect E^i properly. In l_i and r_i, a single event instance is stored, t_i stores a time instant and q_i contains a set of event instances. The symbol $\langle\rangle$ is used to represent a non-occurrence, and we define $\text{start}(\langle\rangle) = \text{end}(\langle\rangle) = -1$ to simplify the algorithm.

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 expressions are static, and known at the time of development, the main loop can be unrolled and the top-level conditionals, as well as all variable indices, can be statically determined. A concrete example of this is given in Figure 4.2.

In the rest of this section we prove that the output of this algorithm corresponds to a valid restriction of $[\![E]\!]$. For this purpose, we need a few auxiliary concepts. First, we formalise the algorithm output by constructing corresponding event streams.

Definition 4.2. For $1 \leq i \leq m$, define

$$\mathcal{A}(i) = \{e \mid e = \text{out}(i, \tau) \wedge e \neq \langle\rangle \wedge \tau \in \mathcal{T}\}$$

where $\text{out}(i, \tau)$ denotes the value of variable a_i after executing the algorithm at times 0 to τ.

We also introduce what can be thought of as a pointwise restriction predicate, and a lemma that captures how it relates to the ordinary restriction policy.

Definition 4.3. For an event instance e, an event stream S and $\tau \in \mathcal{T}$, define $\textit{valid}(e, S, \tau)$ to hold if:

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

Lemma 4.1.1. For event instances $e_0, e_1, e_2, \ldots$ and an event stream S such that $\textit{valid}(e_\tau, S, \tau)$ holds for any $\tau \in \mathcal{T}$, let $S' = \{e_0, e_1, e_2, \ldots\} - \{\langle\rangle\}$. Then $\textit{rem}(S, S')$ 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 $\textit{valid}(e_\tau, S, \tau)$, we must have $e_\tau \neq \langle\rangle$, and thus $e_\tau \in S'$. From the definition of *valid*, we know

```
for i from 1 to m
    if E^i ∈ 𝒫 then
        a_i := the current instance of E^i, or ⟨⟩ if there is none.
    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
    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 = ⟨⟩ or r_i = ⟨⟩ or (a_j = ⟨⟩ and a_k = ⟨⟩) then a_i := ⟨⟩
            else if start(a_k) ≤ start(a_j)
                then a_i := a_j ∪ r_i
                else a_i := l_i ∪ a_k
    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 := ⟨⟩
    if E^i = E^j;E^k then
        a_i := ⟨⟩
        if a_k ≠ ⟨⟩ then
            foreach e in q_i
                if end(e) < start(a_k) and start(a_i) < start(e)
                then a_i := e
            if a_i ≠ ⟨⟩ then a_i := a_k ∪ a_i
        if t_i < start(a_j) then
            q_i := q_i ∪ {a_j}
            t_i := start(a_j)
    if E^i = (E^j)_τ then
        if end(a_j) − start(a_j) ≤ τ then a_i := a_j else a_i := ⟨⟩
```

Figure 4.1: For a given event expression E this algorithm computes an event stream S for which $rem([\![E]\!], S)$ holds. Initially, $t_i = -1$, $l_i = r_i = \langle\rangle$ and $q_i = \emptyset$ for $1 \leq i \leq m$.

a_1 := the current instance of T, or $\langle\rangle$ if there is none.
a_2 := the current instance of P, or $\langle\rangle$ if there is none.
if $a_1 = \langle\rangle$ or ($a_2 \neq \langle\rangle$ and $\mathrm{start}(a_1) \leq \mathrm{start}(a_2)$) then $a_3 := a_2$
    else $a_3 := a_1$
a_4 := the current instance of B, or $\langle\rangle$ if there is none.
if $a_4 \neq \langle\rangle$ and $t_5 < \mathrm{start}(a_4)$ then $t_5 := \mathrm{start}(a_4)$
if $a_3 \neq \langle\rangle$ and $t_5 < \mathrm{start}(a_3)$ then $a_5 := a_3$ else $a_5 := \langle\rangle$

Figure 4.2: Statically simplified algorithm for detecting $(\mathsf{T}\vee\mathsf{P})-\mathsf{B}$. Initially, $t_5 = -1$.

that $\mathrm{start}(s) \leq \mathrm{start}(e_\tau)$. We also have $\mathrm{end}(e_\tau) = \mathrm{end}(s)$, which means that the second requirement in the definition of *rem* is satisfied. Finally, all elements in S' have different end times. Together, this implies that $rem(S, S')$ holds. □

The correctness proof is organised as follows. For each of the six operators, we prove a lemma showing that the operator is correctly detected with respect to the instances detected for its arguments. Finally, these lemmas are combined with Theorem 3.2.8 from the previous chapter to prove the algorithm correct.

Before turning to the operators, some general observations regarding the algorithm can be made. It is straightforward to see that during the ith iteration of the loop only variables with index i are changed, and all variables that are used have indices less than or equal to i, since the subexpressions of E are numbered in bottom-up order. Thus, when proving correctness for subexpression E^i, it is sufficient to consider the ith iteration. The auxiliary predicates defined in this section do not refer to variables of index higher than i, and thus if they hold after the ith iteration, they will hold after iterations $i+1$ to m as well.

4.1.1 Disjunction

The disjunction operator is fairly simple and requires no auxiliary variables. If E^j and E^k occur at the same time, the restriction policy requires that the one with latest start time is selected. When the start times are the same, this implementation gives precedence to the right subexpression.

Lemma 4.1.2. For $1 \le i \le m$ such that $E^i = E^j \vee E^k$, and any $\tau \in \mathcal{T}$, the following holds:

i. $\mathit{valid}(a_i, \mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds after executing the algorithm at time τ.

ii. $\mathit{rem}(\mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$.

Proof.

i. If one or both of a_j and a_k are $\langle\rangle$, then $\mathit{valid}(a_i, \mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ trivially holds after executing the disjunction part of the algorithm. Otherwise, both a_j and a_k belong to $\mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k))$, and thus the one with maximum start time satisfies the condition of *valid*. If the start times are equal, the algorithm selects a_k, which satisfies the condition.

ii. Follows from i and Lemma 4.1.1.

□

4.1.2 Conjunction

For conjunctions, it is necessary to store for each of the two subexpressions the instance with maximum start time so far. This is formalised by the following definition, which holds at the start of time instant τ if l_i and r_i have correct values.

Definition 4.4. For $1 \le i \le m$ such that $E^i = E^j + E^k$, and for $\tau \in \mathcal{T}$, we define $\mathit{constate}(i, \tau)$ to hold if the following holds:

- l_i is an element in $\{e \mid e \in \mathcal{A}(j) \wedge \mathrm{end}(e) < \tau\}$ with maximum start time, or $\langle\rangle$ if that set is empty.
- r_i is an element in $\{e \mid e \in \mathcal{A}(k) \wedge \mathrm{end}(e) < \tau\}$ with maximum start time, or $\langle\rangle$ if that set is empty.

Lemma 4.1.3. For $1 \le i \le m$ such that $E^i = E^j + E^k$, and any $\tau \in \mathcal{T}$, the following holds:

i. $\mathit{constate}(i, \tau)$ holds at the start of time τ.

ii. $\mathit{valid}(a_i, \mathrm{con}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds after executing the algorithm at time τ.

iii. $rem(\text{con}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$.

Proof.

i. $constate(i, 0)$ holds for an initial state where $l_i = r_i = \langle\rangle$. Next, assume that $constate(i, \tau)$ holds at the start of time τ. Then the first conditional in the conjunction part of the algorithm ensures that l_i contains an instance consistent with $constate(i, \tau{+}1)$, after executing the conjunction part. Similarly, the second conditional ensures the correctness of r_i. By induction, the lemma holds for any $\tau \in \mathcal{T}$.

ii. From the proof of i, we know that $constate(i, \tau + 1)$ holds after executing the first two conditionals of the conjunction part. If the guard of the third conditional is satisfied, it trivially follows that there is no instance in $\text{con}(\mathcal{A}(j), \mathcal{A}(k))$ with end time τ, and thus the lemma holds after assigning $\langle\rangle$ to a_i. If the guard is false, we identify three separate cases. For the case when $a_j = \langle\rangle$, we know that $l_i \cup a_k \in \text{con}(\mathcal{A}(j), \mathcal{A}(k))$. Assume the existence of $e \in \text{con}(\mathcal{A}(j), \mathcal{A}(k))$ with $\text{start}(l_i \cup a_k) < \text{start}(e)$ and $\text{end}(e) = \tau$. Then, as $a_j = \langle\rangle$, we must have $e = e' \cup a_k$ with $e' \in \mathcal{A}(j)$, $\text{start}(l_i) < \text{start}(e')$ and $\text{end}(e') < \tau$. This contradicts $constate(i, \tau{+}1)$, and thus no such e' exists which means that $l_i \cup a_k$ is valid. Since $a_j = \langle\rangle$, the inner conditional evaluates to false and $l_i \cup a_k$ is assigned to a_i, meaning that the lemma holds for this case. Similarly, for the case when $a_k = \langle\rangle$ the lemma holds after after assigning $a_j \cup r_i$ to a_i. The third case, when neither a_j nor a_k are $\langle\rangle$, both $a_j \cup r_i$ and $l_i \cup a_k$ belong to $\text{con}(\mathcal{A}(j), \mathcal{A}(k))$. Using the same reasoning as in the previous cases, we have that there can exist no $e \in \text{con}(\mathcal{A}(j), \mathcal{A}(k))$ with $\text{start}(l_i \cup a_k) < \text{start}(e)$, $\text{start}(a_j \cup r_i) < \text{start}(e)$ and $\text{end}(e) = \tau$. If the inner conditional holds, we have $\text{start}(a_k) \leq \text{start}(a_j)$ and by $constate(i, \tau{+}1)$ we also have $\text{start}(a_k) \leq \text{start}(r_i)$. Thus $\text{start}(l_i \cup a_k) \leq \text{start}(a_j \cup r_i)$, and the lemma holds after assigning $a_j \cup r_i$ to a_i. Similarly, if the inner conditional does not hold, the lemma holds after assigning $l_i \cup a_k$ to a_i.

iii. Follows from ii and Lemma 4.1.1.

□

4.1.3 Negation

According to the semantics of the negation operator, an instance of B is an instance of $B-C$ unless it is invalidated by some instance of C occurring within its interval. If the current instance of B is invalidated at all, it is invalidated by the instance of C with maximum start time (of those that have occurred so far). Thus, it is sufficient to store a single start time, since the end time is trivially known to be less than the end time of the current instance of B.

Definition 4.5. For $1 \leq i \leq m$ such that $E^i = E^j - E^k$, and for $\tau \in \mathcal{T}$, we define *negstate*(i, τ) to hold if t_i is the maximum element in the set $\{\text{start}(e) \mid e \in \mathcal{A}(k) \wedge \text{end}(e) < \tau\}$, or -1 if this set is empty.

Lemma 4.1.4. For $1 \leq i \leq m$ such that $E^i = E^j - E^k$, and any $\tau \in \mathcal{T}$, the following holds:

i. *negstate*(i, τ) holds at the start of time τ.

ii. *valid*$(a_i, \text{neg}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds after executing the algorithm at time τ.

iii. *rem*$(\text{neg}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$.

Proof.

i. *negstate*$(i, 0)$ holds for an initial state where $t_i = -1$. Next, assume that *negstate*(i, τ) holds at the start of time τ. Then the first conditional in the negation part of the algorithm ensures that t_i contains the value specified by *negstate*$(i, \tau+1)$ after executing the negation part. By induction, the lemma holds for any $\tau \in \mathcal{T}$.

ii. From the proof of i, we know that *negstate*$(i, \tau+1)$ holds after executing the first conditional of the negation part. If the guard of the second conditional holds, then we have $a_j \neq \langle\rangle$ and thus $a_j \in \mathcal{A}(j)$. According to *negstate*$(i, \tau+1)$ there is no e in $\mathcal{A}(k)$ with $\text{start}(a_j) \leq \text{start}(e)$ and $\text{end}(e) < \text{end}(a_j) = \tau$, and thus $a_j \in \text{neg}(\mathcal{A}(j), \mathcal{A}(k))$. Trivially, since a_j is the only instance in $\mathcal{A}(j)$ with end time τ, we have *valid*$(a_j, \text{neg}(\mathcal{A}(j), \mathcal{A}(k)))$. Thus, the lemma holds after assigning a_j to a_i.

iii. Follows from ii and Lemma 4.1.1.

□

4.1.4 Sequence

The sequence operator requires the most complex algorithm. The reason for this is that in order to detect a sequence $B;C$ correctly, we must store several instances of B. Once C occurs, the start time of that instance determines with which of the stored instances of B it should be combined to form the instance of $B;C$.

Definition 4.6. For $1 \leq i \leq m$ such that $E^i = E^j;E^k$, and for $\tau \in \mathcal{T}$, we define *seqstate*(i, τ) to hold if the following holds:

- t_i is the maximum element in $\{\text{start}(e) \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < \tau\}$, or -1 if this set is empty.
- $q_i = \{e \mid \mathcal{A}(j) \wedge \text{end}(e) < \tau \wedge$
  $\neg\exists e'(e' \in \mathcal{A}(j) \wedge e' \neq e \wedge \text{start}(e) \leq \text{start}(e') \wedge \text{end}(e') \leq \text{end}(e))\}$

Lemma 4.1.5. For $1 \leq i \leq m$ such that $E^i = E^j;E^k$, and any $\tau \in \mathcal{T}$, the following holds:

i. *seqstate*(i, τ) holds at the start of time τ.

ii. *valid*$(a_i, \text{seq}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds after executing the algorithm at time τ.

iii. *rem*$(\text{seq}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$.

Proof.

i. *seqstate*$(i, 0)$ holds for an initial state where $t_i = -1$ and $q_i = \emptyset$. Next, assume that *seqstate*(i, τ) holds at the start of time τ. The first conditional of the sequence part of the algorithm does not change the values of t_i and q_i. If the second conditional holds, t_i is updated to the value specified by *seqstate*$(i, \tau+1)$. Also, by *seqstate*(i, τ), we know that there is no $e \in \mathcal{A}(j)$ with $t_i < \text{start}(e)$ and $\text{end}(e) < \tau$, which implies that *seqstate*$(i, \tau+1)$ holds after adding a_j to q_i. If the second conditional does not hold, no changes are required for *seqstate*$(i, \tau+1)$ to hold. By induction, the lemma holds.

ii. From i, we know that *seqstate*(i, τ) holds at the start of time τ. Consider first the case when $a_k = \langle\rangle$. Then there is no instance in $\text{seq}(\mathcal{A}(j), \mathcal{A}(k))$ with end time τ. Thus, the lemma holds after

assigning $\langle\rangle$ to a_i. In the second case we have $a_k \neq \langle\rangle$. If $a_i = \langle\rangle$ after executing the foreach statement, then there is no instance e in q_i with $\text{end}(e) < \text{start}(a_k)$, and thus by *seqstate*$(i, \tau)$ there is no $e \in \mathcal{A}(j)$ with $\text{end}(e) < \text{start}(a_k)$. This implies that there is no $e' \in \text{seq}(\mathcal{A}(j), \mathcal{A}(k))$ with $\text{end}(e') = \tau$, and thus the lemma holds after assigning $\langle\rangle$ to a_i. If $a_i \neq \langle\rangle$ after the foreach statement, we have $\text{end}(a_i) < \text{start}(a_k)$ and thus $a_i \cup a_k \in \text{seq}(\mathcal{A}(j), \mathcal{A}(k))$. By *seqstate*$(i, \tau)$ we also know that there is no $e' \in \mathcal{A}(j)$ with $\text{start}(a_i) < \text{start}(e')$ and $\text{end}(e') < \text{start}(a_k)$, and thus we have *valid*$(a_i \cup a_k, \text{seq}(\mathcal{A}(j), \mathcal{A}(k))$ and the lemma holds after assigning $a_i \cup a_k$ to a_i.

iii. Follows from ii and Lemma 4.1.1.

□

4.1.5 Temporal Restriction

The temporal restriction is fairly straightforward to implement and requires no auxiliary state variables.

Lemma 4.1.6. For $1 \leq i \leq m$ such that $E^i = (E^j)_{\tau'}$, and any $\tau \in \mathcal{T}$, the following holds:

i. *valid*$(a_i, \text{tim}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds after executing the algorithm at time τ.

ii. *rem*$(\text{tim}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$.

Proof.

i. If $a_j = \langle\rangle$, the lemma holds after assigning $\langle\rangle$ to a_i, which is done in both branches of the conditional. If $a_j \neq \langle\rangle$ and the conditional holds, we have $a_j \in \text{tim}(\mathcal{A}(j), \tau')$. Since a_j is the only instance in $\mathcal{A}(j)$ with end time τ, it follows that the lemma holds after assigning a_j to a_i. If the conditional is false, there is no instance in $\text{tim}(\mathcal{A}(j), \tau')$ with end time τ, so the lemma holds after assigning $\langle\rangle$ to a_i.

ii. Follows from i and Lemma 4.1.1.

□

4.1.6 Putting it all Together

The following theorem establishes the correctness of the algorithm. For each subexpression E^i, including E itself, the detected instances correspond to a valid restriction of $[\![E^i]\!]$.

Theorem 4.1.1. For any i such that $1 \leq i \leq m$, $\mathit{rem}([\![E^i]\!], \mathcal{A}(i))$ holds.

Proof. As the base case, consider $E^i \in \mathcal{P}$. Then $\mathcal{A}(i) = [\![E^i]\!]$ under the assumption that the interpretation $\mathcal{I}$ correctly represents the real-world scenario, and thus $\mathit{rem}([\![E^i]\!], \mathcal{A}(i))$ holds trivially.

For the inductive case, assume that for some x, $\mathit{rem}([\![E^i]\!], \mathcal{A}(i))$ holds for any $1 \leq i < x$. If $E^x = E^j \vee E^k$, then according to Lemma 4.1.2 we have $\mathit{rem}(\mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(x))$. Since the subexpressions are numbered bottom-up, we have $j < x$ and $k < x$, so by assumption $\mathit{rem}([\![E^j]\!], \mathcal{A}(j))$ and $\mathit{rem}([\![E^k]\!], \mathcal{A}(k))$ holds. According to Theorem 3.2.8, this implies $\mathit{rem}(\mathrm{dis}([\![E^j]\!], [\![E^k]\!]), \mathcal{A}(x))$, so $\mathit{rem}([\![E^i]\!], \mathcal{A}(i))$ holds for any $1 \leq i < x+1$. Using the other lemmas in this section, a similar proof can be constructed for each of the other operators as well. By induction, the theorem holds for any $1 \leq i \leq m$. ☐

4.2 Resource Requirements

Instances are not of a fixed size, but an instance of any subexpression of E contains at most $m/2$ primitive instances, one from each identifier occurrence in E. Thus, assuming that the elements in the value domains are of constant size, the size of a single event instance is bounded.

A quick analysis of the algorithm presented in the previous section reveals that each disjunction, conjunction, negation and temporal restriction in the event expression requires a limited amount of storage, and contributes with a limited factor to the computation time of the whole detection algorithm. The storage required for a sequence operator depend on the maximum size of q_i, for which no bound exists in the general case. For an important class of sequence expressions, however, the detection algorithm can be redefined to ensure bounded memory and time.

4.2.1 Memory Complexity

For a sequence $A;B$ where we know that the length of the instances of B never exceeds τ, which can be expressed by the equivalence $B \equiv B_\tau$, this

limits the number of instances of A we need to store in order to detect the sequence correctly. Informally, the start of any instance of B will be at most τ time units back in time, and thus there is no need to store more than one instance of A that ends earlier than this, if we store one with maximum start time. From the instances of A that end later than this point in time, we need to store several, as in the original algorithm version. The following example illustrates this idea.

Example 4.1. A possible scenario during the detection of the event expression $A;B_4$ is depicted in Figure 4.3. Before the current time instant 10, there has been six occurrences of A. If a B_4 instance occurs in the current time instance, and the start time of this instance is 9 or 10, it should be combined with a_6 to form an instance of $A;B_4$. If the start time is 8, it should be combined with a_5 and if it is 7 it should be combined with a_4, etc. Since an instance of B_4 with an end time of 10 must start no earlier than 6, it follows that it must be combined with one of a_3, a_4, a_5 or a_6, and thus there is no need to store a_1 and a_2. Throughout the detection of this expression, all instances of A that ended more than 4 time units ago, except the one with latest start time, can be discarded. ◇

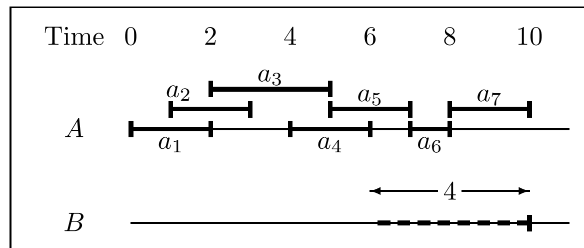


Figure 4.3: Detecting $A;B_4$ with bounded memory.

The improved algorithm for detecting $A;B$ when $B \equiv B_\tau$ with limited memory is presented in Figure 4.4. Here, τ^c denotes the current time instant in which the algorithm is executed.

For this new algorithm, we prove a lemma similar to those in the previous section. The state is similar to the state used for sequences in the original algorithm, but this q_i contains only a suffix of q_i in the

```
a_i := ⟨⟩
if a_k ≠ ⟨⟩ then
    foreach e in q_i
        if end(e) < start(a_k) and start(a_i) < start(e) then a_i := e
    if a_i = ⟨⟩ then a_i := l_i
    if a_i ≠ ⟨⟩ then a_i := a_k ∪ a_i
if t_i < start(a_j) then q_i := q_i ∪ {a_j};  t_i := start(a_j)
foreach e in q_i
    if end(e) ≤ τ^c − τ′ then q_i := q_i − {e};  l_i := e
```

Figure 4.4: Algorithm for $E^i = E^j;E^k$ when $E^k \equiv E^k_{\tau'}$.

original version. From the remaining elements, a single element with maximum start time is stored in l_i.

Definition 4.7. For $1 \leq i \leq m$ such that $E^i = E^j;E^k$ and $E^k \equiv E^k_{\tau'}$, and for $\tau \in \mathcal{T}$, we define *newstate*(i, τ) to hold if the following holds:

- t_i is the maximum element in $\{\text{start}(e) \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < \tau\}$, or -1 if this set is empty.
- $q_i = \{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < \tau \wedge \tau - \tau' \leq \text{end}(e) \wedge$
  $\neg \exists e'(e' \in \mathcal{A}(j) \wedge e' \neq e \wedge \text{start}(e) \leq \text{start}(e') \wedge \text{end}(e') \leq \text{end}(e))\}$
- l_i is an element in $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < \tau - \tau'\}$ with maximum start time, or $\langle\rangle$ if that set is empty.

Lemma 4.2.1. For $1 \leq i \leq m$ such that $E^i = E^j;E^k$ and $E^k \equiv E^k_{\tau'}$, and any $\tau \in \mathcal{T}$, the following holds:

i. *newstate*(i, τ) holds at the start of time τ.

ii. *valid*$(a_i, \text{seq}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds after executing the algorithm at time τ.

iii. *rem*$(\text{seq}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$.

Proof.

i. *newstate*$(i, 0)$ holds for an initial state where $t_i = -1$, $q_i = \emptyset$ and $l_i = \langle\rangle$. Next, assume that *newstate*(i, τ) holds at the start of time

τ. The first conditional only changes a_i, and thus does not affect *newstate*. The second conditional updates q_i and t_i in the same way as in the original sequence algorithm. If the conditional of the last foreach statement holds for e, we actually have $\text{end}(e) = \tau - \tau'$. This means that the conditional holds for at most one element of q_i. The definition of *newstate* requires this e to be removed from q_i in order for $\textit{newstate}(i, \tau+1)$ to hold. By $\textit{newstate}(i, \tau)$, we also have that e fulfills the requirement on l_i as specified by $\textit{newstate}(i, \tau+1)$. By induction, the lemma holds.

ii. From i, we know that $\textit{newstate}(i, \tau)$ holds at the start of time τ.

The case when $a_k = \langle\rangle$ follows the proof for the original sequence. In the case when $a_k \neq \langle\rangle$, if $a_i \neq \langle\rangle$ after executing the first foreach statement, then we have $\textit{valid}(a_i \cup a_k, \text{seq}(\mathcal{A}(j), \mathcal{A}(k))$ as in the proof of the original sequence. If $a_i = \langle\rangle$ after executing the first foreach statement, then there can be no instance e in q_i with $\text{end}(e) < \text{start}(a_k)$. In this case, we assign l_i to a_i. If $l_i \neq \langle\rangle$ we know that $\text{end}(l_i) < \text{start}(a_k)$, and then since l_i has the value specified by $\textit{newstate}(i, \tau+1)$, we have that $\textit{valid}(l_i \cup a_k, \text{seq}(\mathcal{A}(j), \mathcal{A}(k)))$ holds.

Thus, arriving at the next conditional we know that either $a_i = \langle\rangle$ and there is no instance e in $\mathcal{A}(j)$ with $\text{end}(e) < \text{start}(a_k)$, or that $\textit{valid}(a_i \cup a_k, \text{seq}(\mathcal{A}(j), \mathcal{A}(k)))$ holds. Thus, the lemma holds when the second foreach statement is reached, and since a_i is not assigned any value in the remaining statements, the lemma holds at the end as well.

iii. Follows from ii and Lemma 4.1.1.

□

From the definition of *newstate* it follows that for a sequence $A;B$ where $B \equiv B_\tau$, the size of q_i never exceeds τ at the start of each time instant. Since at most one instance is added to q_i during a single execution of the algorithm, this means that the storage requirement is limited for such sequences.

For large values of τ, for example in systems with a fine granularity timebase, this resource bound might not be sufficient in practice. Also, for $A;B$ the bound is a large overapproximation unless A occurs very frequently. If we have information about the minimum separation time

of primitive events (i.e., the minimum time between two consecutive occurrences of the same event), we can derive the maximum number of occurrences of A during any time interval of length τ.

Definition 4.8. The minimum separation time of primitive events is represented by the function $\text{minsep} : \mathcal{P} \rightarrow \mathbb{Z}^+$. An interpretation $\mathcal{I}$ is *consistent* with minsep if the following holds. For any $p \in \mathcal{P}$, if $\langle p, \upsilon, \tau \rangle \in \mathcal{I}(p)$ and $\langle p, \upsilon', \tau' \rangle \in \mathcal{I}(p)$ with $\tau < \tau'$ then $\text{minsep}(p) \leq \tau' - \tau$.

Based on the minimum separation time we could compute minimum separation times for composite events, but for disjunction and conjunction the minimum separation is 1 regardless of the constituent events. Instead, we derive the maximum number of times an event can occur in any interval of a given size.

Definition 4.9. The function occ is defined as follows:

$$\begin{array}{lcl}
\text{occ}(A,\tau) & = & \lceil \tau/\text{minsep}(A) \rceil \qquad \text{if } A \in \mathcal{P} \\
\text{occ}(A \vee B,\tau) & = & \min(\tau,\ \text{occ}(A,\tau) + occ(B,\tau)) \\
\text{occ}(A + B,\tau) & = & \min(\tau,\ \text{occ}(A,\tau) + occ(B,\tau)) \\
\text{occ}(A;B,\tau) & = & \text{occ}(B,\tau) \\
\text{occ}(A - B,\tau) & = & \text{occ}(A,\tau) \\
\text{occ}(A_{\tau'},\tau) & = & \text{occ}(A,\tau)
\end{array}$$

This function represents the maximum number of A instances with different end times that can occur in an interval of length τ, as the following theorem states.

Theorem 4.2.1. For an event expression A, time instants $\tau, \tau' \in \mathcal{T}$ and an interpretation $\mathcal{I}$ that is consistent with minsep, the following holds:

$$|\{\text{end}(a) \mid a \in [\![A]\!] \wedge \tau' \leq \text{end}(a) \wedge \text{end}(a) < \tau' + \tau\}| \ \leq \ \text{occ}(A,\tau)$$

Proof. Proof by induction. For the base case when $A \in \mathcal{P}$, we note that the maximum number of instances in an interval is achieved if instances occur at $\text{minsep}(A)$ distance. Then, exactly $\text{occ}(A,\tau)$ instances fit into the interval. For the inductive case, assume that the property holds for event expressions A and B. There are only τ time points in the interval, so the number of end times can not be larger than τ. This justifies the minimum construct for disjunction and conjunction.

Every instance of $A \vee B$ has an end time equal to the end time of an instance of A or B, so the number of distinct end times is limited by the

sum of distinct end times for A and B for this interval. According to the assumption, this is not larger than $\text{occ}(A, \tau) + occ(B, \tau)$. Since all instances of $A+B$ must have the same end time as some instance from A or from B, the reasoning for disjunction applies to conjunctions as well.

An instance of $A;B$ must have the same end time as an instance of B, and by assumption there are no more than $occ(B, \tau)$ distinct instances of B in the interval. Similarly, all instances of $A-B$, and all instances of $A_{\tau'}$, must have the same end time as an instance from A. By induction, the theorem holds for any expression. □

Corollary 4.2.1. In the improved sequence algorithm for $E^i = E^j;E^k$ when $E^k \equiv E^k_{\tau'}$, the size of q_i never exceeds $\text{occ}(E^j, \tau')$.

Proof. All elements in q_i belong to $[\![E^j]\!]$, and according to the constraint on q_i defined by *newstate* they all have end times in an interval of length τ'. Then, the corollary follows directly from the theorem. □

For some applications it can be beneficial to provide maximum occurrence values for some primitive events in addition to the minimum separation time. This is especially useful for "bursty" events that can occur a few times with high frequency but then do not occur for a long time. Extending the definition above to take this kind of information into account is relatively straightforward.

To formally identify the class of expressions that can be detected with bounded memory, we formulate the following theorem.

Theorem 4.2.2. Let E be an event expression such that for any subexpression of E on the form $A;B$, we have $B \equiv B_\tau$ for some $\tau \in \mathcal{T}$. Then, E can be correctly detected with limited memory.

Proof. Trivially, the memory needed to store information about the past is limited for all operators except sequence. In the algorithm for detecting this specific type of sequences, the size of q_i is limited. Thus, the total storage requirement is limited. □

This theorem does not provide means for syntactically determining if an expression can be detected with limited memory. Chapter 5 describes how the laws of temporal restriction can be used to transform expressions to meet the criteria under which limited resource requirements are guaranteed. In addition, sufficient criteria are presented to syntactically identify expressions that require only limited memory when transformation is applied.

4.2.2 Time Complexity

As a result of instances not having a fixed size, assigning an instance to a variable might not be a constant operation, but proportional to the instance size. Thus, each operator contributes with at least a factor m to the time complexity of the whole algorithm. For sequences, a straightforward representation of the q_i variables gives a linear time complexity for finding the best matching instance, with respect to the size limit of that q_i variable. This gives a a total complexity of $O(mn')$, where m is the number of subexpressions in E, $n' = \max(m, n)$ and n is the maximum size limit of the q_i variables.

When the maximum size of q_i is known, a more elaborate implementation is possible. We keep q_i sorted with respect to end times, and since *newstate* ensures that q_i does not contain instances that are fully overlapping, this means that they will be sorted with respect to start times as well. Since an instance that is added to q_i has a later start time than the instances already in q_i, and elements are removed when they become too old, q_i will behave like a first-in first-out queue.

Consequently, keeping q_i updated can be done in constant time. However, when an instance of B occurs we need to find the best matching instance in q_i efficiently, i.e., the instance with latest start time among those that end before the start time of the B instance. Since q_i is sorted with respect to both start and end times, this can be implemented as a straightforward binary search if the implementation of q_i allows constant access to arbitrary elements.

To allow this, we base the implementation of q_i on a static array, and use two variables to mark the part of this array that currently contain q_i. When elements are added and removed, these variables are increased accordingly, and when the end of the array is reached they simply continue at the beginning. Since the size limit of q_i is known, using the same size for the array ensures that there is always room for q_i in the array, i.e., that the start marker will not overtake the end marker. The following example illustrates the implementation of the q_i variables

Example 4.2. Figure 4.5 describes how instances of A are stored in the q_i variable during the detection of $A;B_4$, assuming that no information about minimum separation is available. The figure also shows how l_i is updated with the element most recently removed from q_i. ◇

The improved sequence algorithm is presented in Figure 4.6. The

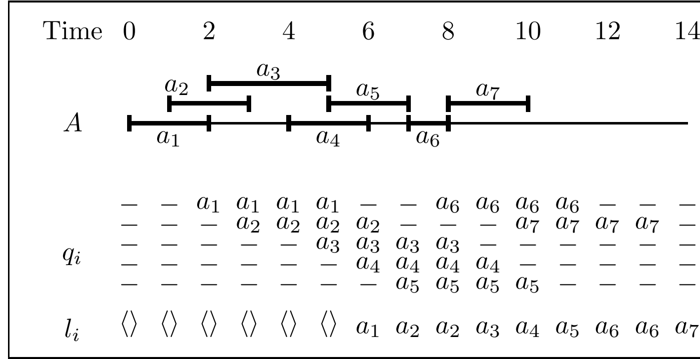


Figure 4.5: An example of how instances of A are stored in q_i and l_i during the detection of $A;B_4$.

algorithm uses two indexed integer variables s_i and e_i, and qs, qe and qm which are temporary variables that can be shared by all sequence operators in the expression. The variable s_i marks the start of the active part of q_i, where new instances are added, and e_i marks the end of the active part. In the initial state we have $s_i = e_i$, representing that q_i is empty. The temporary variables mark the start, end and middle of a subsection of the active part during the binary search. Accessing the xth element of the array q_i is denoted $q_i[x]$. Using this implementation, the total complexity of detecting E is $O(mn'')$, where m is the number of subexpressions in E, $n'' = \max(m, \log n)$ and n is the maximum size limit of the q_i variables.

```
a_i := ⟨⟩
if a_k ≠ ⟨⟩ then
    if s_i ≠ e_i and end(q_i[e_i]) < start(a_k) then
        qe := e_i
        qs := s_i − 1 mod o
        while qe ≠ qs
            qm := qe + ⌈((qs − qe) mod o)/2⌉ mod o
            if end(q_i[qm]) < start(a_k) then qe := qm
                else qs := qm − 1 mod o
        a_i := q_i[qe]
    if a_i = ⟨⟩ then a_i := l_i
    if a_i ≠ ⟨⟩ then a_i := a_k ∪ a_i
if t_i < start(a_j) then
    q_i[s_i] := a_j
    s_i := s_i + 1 mod o
    t_i := start(a_j)
if s_i ≠ e_i and end(q_i[e_i]) ≤ τ^c − τ' then
    l_i := q_i[e_i]
    e_i := e_i + 1 mod o
```

Figure 4.6: Improved algorithm for $E^i = E^j;E^k$ when $E^k \equiv E^k_{\tau'}$. Initially $t_i = -1$, $s_i = e_i = 0$ and $l_i = \langle\rangle$. For brevity, o denotes $\mathrm{occ}(E^j, \tau'+1)$.

Chapter 5

Expression Transformation

This chapter describes how an event expression can be automatically transformed into an expression that can often be detected more efficiently, but has the same meaning. The transformation algorithm is based on the algebraic laws describing how temporal restrictions can be propagated through an expression, presented in Section 3.2.

To simplify the presentation, we extend the algebra syntax with two constructs. The symbol ∞ is added to the temporal domain to allow temporally restricted and unrestricted expressions to be treated uniformly. Formally, we define $A_\infty = A$. Since the improved sequence algorithm is defined for sequences $A;B$ where $B \equiv B_\tau$ for some τ, we introduce the notation $A;_\tau B$ to label sequences with this information.

5.1 The Transformation Algorithm

The transformation algorithm is based on a recursive function that takes an expression and a time instant as input, and returns the transformed expression and a time instant. This function is defined in Figure 5.1. Informally, the input time instant represents a temporal restriction that can be applied to the current subexpression without changing the meaning of the expression as a whole. For example, if the whole expression is on the form $(A \vee B)_2$, changing A into A_2 changes the meaning of this subexpression, but the meaning of the whole expression remains unchanged. The returned time instant represents a temporal restriction that can be applied to the transformed expression without changing its

$$\text{transform}(A, \tau) = \langle A, 0 \rangle \qquad \text{if } A \in \mathcal{P}$$

$$\begin{aligned} \text{transform}(A \vee B, \tau) &= \langle A' \vee B', \tau' \rangle \\ \text{where } \langle A', \tau_a \rangle &= \text{transform}(A, \tau) \\ \langle B', \tau_b \rangle &= \text{transform}(B, \tau) \\ \tau' &= \max(\tau_a, \tau_b) \end{aligned}$$

$$\begin{aligned} \text{transform}(A + B, \tau) &= \langle A' + B', \infty \rangle \\ \text{where } \langle A', \tau_a \rangle &= \text{transform}(A, \tau) \\ \langle B', \tau_b \rangle &= \text{transform}(B, \tau) \end{aligned}$$

$$\begin{aligned} \text{transform}(A - B, \tau) &= \langle A' - B', \tau_a \rangle \\ \text{where } \langle A', \tau_a \rangle &= \text{transform}(A, \tau) \\ \tau' &= \min(\tau_a, \tau) \\ \langle B', \tau_b \rangle &= \text{transform}(B, \tau') \end{aligned}$$

$$\begin{aligned} \text{transform}(A ; B, \tau) &= \begin{cases} \langle A' ;_{\tau_b} B', \infty \rangle & \text{if } \tau_b \leq \tau \\ \langle A' ;_{\tau} (B'_{\tau}), \infty \rangle & \text{if } \tau < \tau_b \end{cases} \\ \text{where } \langle A', \tau_a \rangle &= \text{transform}(A, \tau) \\ \langle B', \tau_b \rangle &= \text{transform}(B, \tau) \end{aligned}$$

$$\begin{aligned} \text{transform}(A_{\tau'}, \tau) &= \begin{cases} \langle A', \tau_a \rangle & \text{if } \tau_a \leq \tau'' \\ \langle A'_{\tau''}, \tau'' \rangle & \text{if } \tau'' < \tau_a \end{cases} \\ \text{where } \langle A', \tau_a \rangle &= \text{transform}(A, \tau'') \\ t'' &= \min(\tau, \tau') \end{aligned}$$

Figure 5.1: The recursive transformation function.

meaning.

This informal description is formalised in the following lemma, which states that the transformation function preserves the semantics of the original expression when called properly. It also defines the meaning of the returned time.

Lemma 5.1.1. For an event expression E and a time instant $\tau \in \mathcal{T}$, if transform$(E, \tau) = \langle E', \tau' \rangle$, then $E_\tau \equiv E'_\tau$ and $E' \equiv E'_{\tau'}$.

Proof. We prove the lemma by structural induction. In the proof, $\equiv^{22}$ denotes that the equivalence follows from law 22 in Theorem 3.2.4, and $\equiv^a$ indicates that the equivalence follows from the assumptions.

For the base case, when $E \in \mathcal{P}$, we note that the lemma holds for transform$(E, \tau) = \langle E, 0 \rangle$ since $E \equiv^{22} E_0$. For the inductive case we assume that the lemma holds for the subexpressions of E, and consider each operator separately:

Disjunction For $E = A \vee B$, we have $A_\tau \equiv A'_\tau$, $A' \equiv A'_{\tau_a}$, $B_\tau \equiv B'_\tau$ and $B' \equiv B'_{\tau_b}$ by assumption. Then, we have
$(A \vee B)_\tau \equiv^{24} A_\tau \vee B_\tau \equiv^a A'_\tau \vee B'_\tau \equiv^{24} (A' \vee B')_\tau$.
Next, let $\tau' = \max(\tau_a, \tau_b)$. Then $A' \vee B' \equiv^a A'_{\tau_a} \vee B'_{\tau_b} \equiv^{33}$
$(A'_{\tau_a} \vee B'_{\tau_b})_{\tau'} \equiv^a (A' \vee B')_{\tau'}$.
Thus, we have $(A \vee B)_\tau \equiv (A' \vee B')_\tau$ and $A' \vee B' \equiv (A' \vee B')_{\tau'}$, which means that the lemma holds for transform$(A \vee B, \tau) = \langle A' \vee B', \tau' \rangle$.

Conjunction For $E = A + B$, we have $A_\tau \equiv A'_\tau$, $A' \equiv A'_{\tau_a}$, $B_\tau \equiv B'_\tau$ and $B' \equiv B'_{\tau_b}$ by assumption. Then, we have
$(A + B)_\tau \equiv^{25,34} (A_\tau + B_\tau)_\tau \equiv^a (A'_\tau + B'_\tau)_\tau \equiv^{25,34} (A' + B')_\tau$
and by definition $(A' + B')_\infty \equiv (A' + B')$ so the lemma holds for transform$(A + B, \tau) = \langle A' + B', \infty \rangle$.

Negation For $E = A - B$, we assume $A_\tau \equiv A'_\tau$, $A' \equiv A'_{\tau_a}$, $B_{\tau'} \equiv B'_{\tau'}$ and $B' \equiv B'_{\tau_b}$ where $\tau' = \min(\tau_a, \tau)$. Then, we have
$(A - B)_\tau \equiv^{26} A_\tau - B \equiv^a A'_\tau - B \equiv^a (A'_{\tau_a})_\tau - B \equiv^{23} A'_{\tau'} - B \equiv^{26}$
$(A' - B)_{\tau'} \equiv^{27} (A' - B_{\tau'})_{\tau'} \equiv^a (A' - B'_{\tau'})_{\tau'} \equiv^{27} (A' - B')_{\tau'}$ and
$(A' - B')_{\tau_a} \equiv^{26} (A'_{\tau_a} - B') \equiv^a (A' - B')$ so the lemma holds for transform$(A - B, \tau) = \langle A' - B', \tau_a \rangle$.

Sequence For $E = A;B$, we have $A_\tau \equiv A'_\tau$, $A' \equiv A'_{\tau_a}$, $B_\tau \equiv B'_\tau$ and $B' \equiv B'_{\tau_b}$ by assumption. We consider the two cases separately.
If $\tau_b \leq \tau$, then $(A - B)_\tau \equiv^{28,29} (A_\tau - B_\tau)_\tau \equiv^a (A'_\tau - B'_\tau)_\tau \equiv^{28,29}$

$(A'-B')_\tau$ and trivially $(A'-B')_\infty \equiv A'-B'$. Thus, the lemma holds for transform$(A;B,\tau) = \langle A';B',\infty\rangle$ in this case.
If $\tau < \tau_b$, then $(A-B)_\tau \equiv^{28,29} (A_\tau - B_\tau)_\tau \equiv^a (A'_\tau - B'_\tau)_\tau \equiv^{28}$ $(A'-B'_\tau)_\tau$ and trivially $(A'-(B'_\tau))_\infty \equiv (A'-(B'_\tau))$. Thus, the lemma holds for transform$(A;B,\tau) = \langle A';(B'_\tau),\infty\rangle$ in this case.
The labelling has no impact on the semantics of the expression. The fact that sequences are correctly labelled is addressed by Theorem 5.1.1 below.

Temporal For $E = A_{\tau'}$, we have $A_{\tau''} \equiv A'_{\tau''}$ and $A' \equiv A'_{\tau_a}$ by assumption. Let $t'' = \min(\tau,\tau')$, and consider the two cases:
If $\tau_a \le \tau''$, then $\tau_a \le \tau'$ and we have $(A_{\tau'})_\tau \equiv^{30} (A_\tau)_{\tau'} \equiv^a$ $(A'_\tau)_{\tau'} \equiv^{30} (A'_{\tau'})_\tau \equiv^a ((A'_{\tau_a})_{\tau'})_\tau \equiv^{23} (A'_{\min(\tau_a,\tau')})_\tau \equiv (A'_{\tau_a})_\tau \equiv^a$ A'_τ and by assumption $A'_{\tau_a} \equiv A'$. Thus, the lemma holds for transform$(A_{\tau'},\tau) = \langle A',\tau_a\rangle$ in this case.
If $\tau'' < \tau_a$, then we have $(A_{\tau'})_\tau \equiv^{23} ((A_{\tau'})_\tau)_\tau \equiv^{23} (A_{\min(\tau',\tau)})_\tau \equiv$ $(A_{\tau''})_\tau \equiv^a (A'_{\tau''})_\tau$ and $(A'_{\tau''})_{\tau''} \equiv^{23} A'_{\tau''}$ so the lemma holds for transform$(A_{\tau'},\tau) = \langle A'_{\tau''},\tau''\rangle$ in this case.

By induction, the lemma holds for any event expression. ☐

Finally, this lemma is used to prove that the transformation preserves the semantic meaning of the expression. Also, we identify the class of expressions that can be correctly detected with limited memory when the transformation is applied.

Theorem 5.1.1. For an event expression E, let transform$(E,\infty) = \langle E',\tau'\rangle$. Then, the following holds:

i) $E \equiv E'$.

ii) For any subexpression of E' on the form $A;_\tau B$, $B \equiv B_\tau$ holds.

iii) If $\tau \neq \infty$ for every sequence $A;_\tau B$ in E', then E can be correctly detected with limited memory.

Proof.

i) Follows trivially from Lemma 5.1.1.

ii) A subexpression in E' on the form $A;_\tau B$ was created by the sequence part of the transformation algorithm, which has two cases.

In the first case, $\langle B, \tau\rangle$ was the result of a call to transform, which according to Lemma 5.1.1 implies that $B \equiv B_\tau$ In the second case, $B = (B')_\tau$, which trivially means that $B \equiv B_\tau$.

iii) Trivially, all sequences in E' are on the extended form $A;_\tau B$, and according to *ii)* they satisfy the condition for using the specialised sequence algorithm presented in Section 4.2.1.

$\square$

Example 5.1. As a first example, we consider the expression $\mathsf{P}_2;(\mathsf{B}\vee\mathsf{T})$. The calls to the transformation function, and the returned tuples are depicted in Figure 5.2. We see that the temporal restriction can be removed, since instances of P are guaranteed to be instantaneous. Also, the sequence is labelled with information specifying that $\mathsf{B}\vee\mathsf{T} \equiv (\mathsf{B}\vee\mathsf{T})_0$, and thus the size of the corresponding q_i variable will not exceed 1 during the detection. $\diamond$

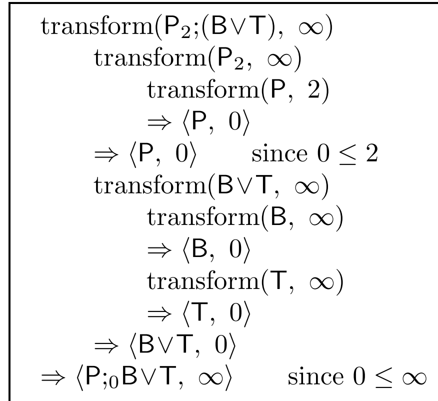


Figure 5.2: Transformation of $\mathsf{P}_2;(\mathsf{B}\vee\mathsf{T})$.

Example 5.2. Next, consider the somewhat more complex expression $(\mathsf{B};\mathsf{B})_2-(\mathsf{P};(\mathsf{P}+\mathsf{T}))$. The transformation, shown in Figure 5.3, illustrates two important aspects of the algorithm. If analysed separately, the second argument of the negation is not detectable with limited resources,

even when the transformation algorithm is applied. However, the negation allows the algorithm to propagate time restriction information from the first argument to the second, where it is used to limit the sequence. The sequence in the second argument to the negation exemplifies that it is sometimes necessary to add explicit time restrictions. According to the information passed down from earlier analysis, there is no need to detect instances of $\mathsf{P}+\mathsf{T}$ that are longer than 2, but transforming this expression yields the information that the instances detected might be of arbitrary length. In order to limit the sequence, an explicit temporal restriction is added. ◇

$$
\begin{array}{l}
\text{transform}((\mathsf{B};\mathsf{B})_2 - (\mathsf{P};(\mathsf{P}+\mathsf{T})),\ \infty) \\
\quad \text{transform}((\mathsf{B};\mathsf{B})_2,\ \infty) \\
\quad\quad \text{transform}(\mathsf{B};\mathsf{B},\ 2) \\
\quad\quad\quad \text{transform}(\mathsf{B},\ 2) \\
\quad\quad\quad \Rightarrow \langle \mathsf{B},\ 0 \rangle \\
\quad\quad\quad \text{transform}(\mathsf{B},\ 2) \\
\quad\quad\quad \Rightarrow \langle \mathsf{B},\ 0 \rangle \\
\quad\quad \Rightarrow \langle \mathsf{B};_0\mathsf{B},\ \infty \rangle \qquad \text{since } 0 \leq 2 \\
\quad \Rightarrow \langle (\mathsf{B};_0\mathsf{B})_2,\ 2 \rangle \qquad \text{since } \min(\infty, 2) < \infty \\
\quad \text{transform}(\mathsf{P};(\mathsf{P}+\mathsf{T}),\ 2) \\
\quad\quad \text{transform}(\mathsf{P},\ 2) \\
\quad\quad \Rightarrow \langle \mathsf{P},\ 0 \rangle \\
\quad\quad \text{transform}(\mathsf{P}+\mathsf{T},\ 2) \\
\quad\quad\quad \text{transform}(\mathsf{P},\ 2) \\
\quad\quad\quad \Rightarrow \langle \mathsf{P},\ 0 \rangle \\
\quad\quad\quad \text{transform}(\mathsf{T},\ 2) \\
\quad\quad\quad \Rightarrow \langle \mathsf{T},\ 0 \rangle \\
\quad\quad \Rightarrow \langle \mathsf{P}+\mathsf{T},\ \infty \rangle \\
\quad \Rightarrow \langle \mathsf{P};_2(\mathsf{P}+\mathsf{T})_2,\ \infty \rangle \qquad \text{since } 2 < \infty \\
\Rightarrow \langle (\mathsf{B};_0\mathsf{B})_2 - (\mathsf{P};_2(\mathsf{P}+\mathsf{T})_2),\ 2 \rangle
\end{array}
$$

Figure 5.3: Transformation of $(\mathsf{B};\mathsf{B})_2 - (\mathsf{P};(\mathsf{P}+\mathsf{T}))$.

Finally, we present sufficient criteria under which an event expression is guaranteed to be detectable with limited resources after applying the transformation algorithm.

Theorem 5.1.2. If at least one of the following criteria holds for every subexpression in E on the form $A;B$, then E can be detected with limited memory after applying the transformation algorithm.

a) $A;B$ occurs inside a temporal restriction, or

b) B does not contain any sequence or conjunction operators.

Proof.

a) From the transformation algorithm it follows that τ is finite in the recursive call transform($A;B,\tau$), since this holds for any subexpression of a temporal restriction. Thus, for both cases in the sequence part of the transformation algorithm, the resulting sequence label is finite.

b) In this case, transform(B,τ) will return a pair $\langle B',\tau_b\rangle$ where τ_b is finite, since infinite return time values are only introduced by sequences and conjunctions. Thus, both cases in the sequence part of the transformation algorithm produce a sequence with finite label.

According to Theorem 5.1.1 this ensures that E can be detected with limited memory. □

Note that these are sufficient but not necessary criteria for limited resource requirements. Some expressions are detectable with limited memory without meeting these conditions, for example the expression in Example 5.2.

5.2 Experiments

We have conducted two experiments to evaluate the transformation algorithm. Expressions containing N operators were created randomly, with equal probability for the five operators to occur. For temporal restrictions, random time values were generated uniformly between 0 and a given limit T. Each configuration is represented by 10,000 expressions.

5.2.1 Experiment 1

The first experiment investigates the class of expressions that can be detected with limited resources when the transformation algorithm is applied. For each randomly generated expression we apply the transformation algorithm and determine if the resulting expression belongs to the class and, if it does, the maximum storage requirement in number of instances. First, the experiment is performed with primitive events given a minsep value of 1, representing that no information about minimum separation is available. Then, the experiment is repeated with half of the primitive events having minsep values uniformly generated between 2 and 50.

T	N	limited memory	25%	50%	75%	average memory
100	5	85%	11	13	16	14.7
100	10	74%	22	27	∞	34.4
100	20	61%	49	121	∞	79.0
100	30	51%	101	232	∞	126.4
300	5	86%	11	13	16	20.4
300	10	73%	22	27	∞	55.3
300	20	60%	50	269	∞	143.5
300	30	52%	158	671	∞	236.0

Table 5.1: Summary of Experiment 1a. All primitive events have a minsep value of 1.

The results are presented in Appendix A but a summary is given in Table 5.1 and Table 5.2. For each configuration, they present the percentage of expressions for which resource requirements are limited. The following three columns indicate the 25th, 50th and 75th percentiles. For example, a value of 11 in the first of these columns means that 25% of the expressions require 11 or fewer instances to be stored. The last column contains the average storage size for expressions with limited resources.

The experiment indicates that the class of expressions that can be detected with limited resources is fairly large. As anticipated, the percentage does not depend on the range of time variables or minsep information. Furthermore, the average storage requirements are reasonably low for all configurations, and does not increase dramatically when the

T	N	limited memory	25%	50%	75%	average memory
100	5	85%	11	13	16	13.6
100	10	74%	22	26	∞	30.3
100	20	60%	48	101	∞	68.0
100	30	52%	83	244	∞	108.9
300	5	85%	11	13	16	16.8
300	10	73%	22	27	∞	46.0
300	20	60%	49	222	∞	116.0
300	30	52%	112	542	∞	190.5

Table 5.2: Summary of Experiment 1b. Half of the primitive events have minsep values uniformly generated between 2 and 50.

time value range is increased from 100 to 300. Introducing information about minimum separation for some primitive events improved the storage size by roughly 20% for the higher time value limit.

5.2.2 Experiment 2

In earlier versions of the algebra, limited resource requirements were ensured by only providing a temporally restricted sequence operator, corresponding to a sequence on the form $(A;B)_\tau$ in the algebra presented in this thesis [11]. Since such an expression is equivalent to $(A;(B_\tau))_\tau$, the specialised sequence algorithm can be applied.

This experiment is restricted to expressions where all sequences are on the form $(A;B)_\tau$, which ensures that the detection can be performed with limited resources. For this type of expressions, we investigate the efficiency gained by introducing the transformation algorithm. Summaries of the results are presented in Table 5.3 and Table 5.4. For each configuration, the tables present three percentiles like in the previous experiment, and the average storage requirement, before and after the transformation algorithm is applied. For additional results, see Appendix A.

The experiment shows that for this class of expressions, syntactically constrained to ensure that the expressions are detectable with limited resources, the average storage size is significantly reduced when the transformation algorithm is applied.

T	N	before transformation 25%	50%	75%	average	after transformation 25%	50%	75%	average
100	5	13	33	81	48.9	9	11	14	16.0
100	10	53	99	141	102.6	20	33	35	34.6
100	20	145	203	262	208.2	43	56	93	71.9
100	30	239	310	386	316.1	69	95	136	108.1
300	5	13	74	213	120.0	9	11	13	25.7
300	10	111	246	366	256.8	20	24	71	61.8
300	20	342	510	696	528.2	43	85	191	132.0
300	30	571	781	1002	799.6	76	158	276	197.0

Table 5.3: Summary of Experiment 2a. All primitive events have a minsep value of 1.

T	N	before transformation 25%	50%	75%	average	after transformation 25%	50%	75%	average
100	5	12	16	49	34.0	9	11	13	14.0
100	10	28	61	106	72.5	19	23	28	30.7
100	20	95	141	194	149.0	41	49	66	63.3
100	30	162	218	280	225.4	65	80	116	95.1
300	5	13	18	119	76.0	9	11	13	19.4
300	10	34	132	267	168.0	19	23	32	49.4
300	20	195	328	485	353.1	42	54	144	106.6
300	30	353	518	714	544.9	68	119	226	163.7

Table 5.4: Summary of Experiment 2b. Half of the primitive events have minsep values uniformly generated between 2 and 50.

Chapter 6

Prototype Implementation

This chapter describes an event triggered prototype implemented in Java, where the algebra is integrated with the existing event framework in the Abstract Window Toolkit (AWT) used for graphical user interface programming.

The detection algorithm presented in Section 4.1 can be directly used as an implementation of the algebra for time triggered reactive systems. Each time instant, the current primitive events are processed and the detected instances of composite events are handed over to the reactive part of the system. When designing real-time systems, the time triggered approach is sometimes considered overly costly, for example in applications where the expected rate of primitive event occurrences is low, and in systems with soft real-time demands. An alternative is to use an event triggered approach where the execution is fully driven by the arrival of events. This results in a lower overhead cost during periods of low activity, at the cost of making it more difficult to guarantee a timely behaviour in a worst case scenario.

The chapter is not intended to describe how to best design an implementation of the algebra in a resource-conscious event triggered system. Rather, the prototype provides a possibility to test the algebra in practice and to highlight some of the issues that are related to the event triggered setting.

6.1 Design Decisions

Integrating the algebra with the existing AWT event handling provides a straightforward way to use the algebra in an application. Primitive events can be tied to events in a graphical interface, such as a button being pressed. An event expression is treated in the same way as a graphical component that produces events, meaning that objects can register that they want to be informed when the event defined by an expression occurs, just like they register with, for example, a button.

The AWT event mechanisms are dynamic, so event producers and consumers can be created, deleted and redefined at runtime. To achieve a seamless integration of the algebra, we want event expressions to exhibit the same high level of dynamicity. An application can construct new composite event expressions from existing expressions or subexpressions at runtime. For example, if A is defined as the expression $\mathsf{P} \vee \mathsf{T}$, then A' can later be defined as $A + \mathsf{B}$. This affects how the transformation algorithm can be applied, as the transformation of a subexpression depends on the expression in which it occurs. Currently, the transformation algorithm is not part of the prototype implementation, and it is assumed that the application performs transformations before constructing the expressions, if necessary. In a sharp implementation, the possibility to dynamically construct composite expressions from existing ones might be restricted to allow the transformation algorithm to be automatically applied.

An expression is represented by objects corresponding to the individual operators, and there is no central object governing each expression. As a result, the way in which the handling of an occurrence propagates through the expression can not be based on global information about the whole expression. This requires a rather intricate communication scheme, as described in the next section.

The implementation is limited to the Java subset defined by J2ME Personal Profile, a set of Java APIs that supports resource-constrained devices [30].

6.2 System Description

A class diagram describing the main classes is depicted in Figure 6.1. Time is measured in milliseconds, following the Java standard. To keep resource requirements low, primitive events can be given minsep val-

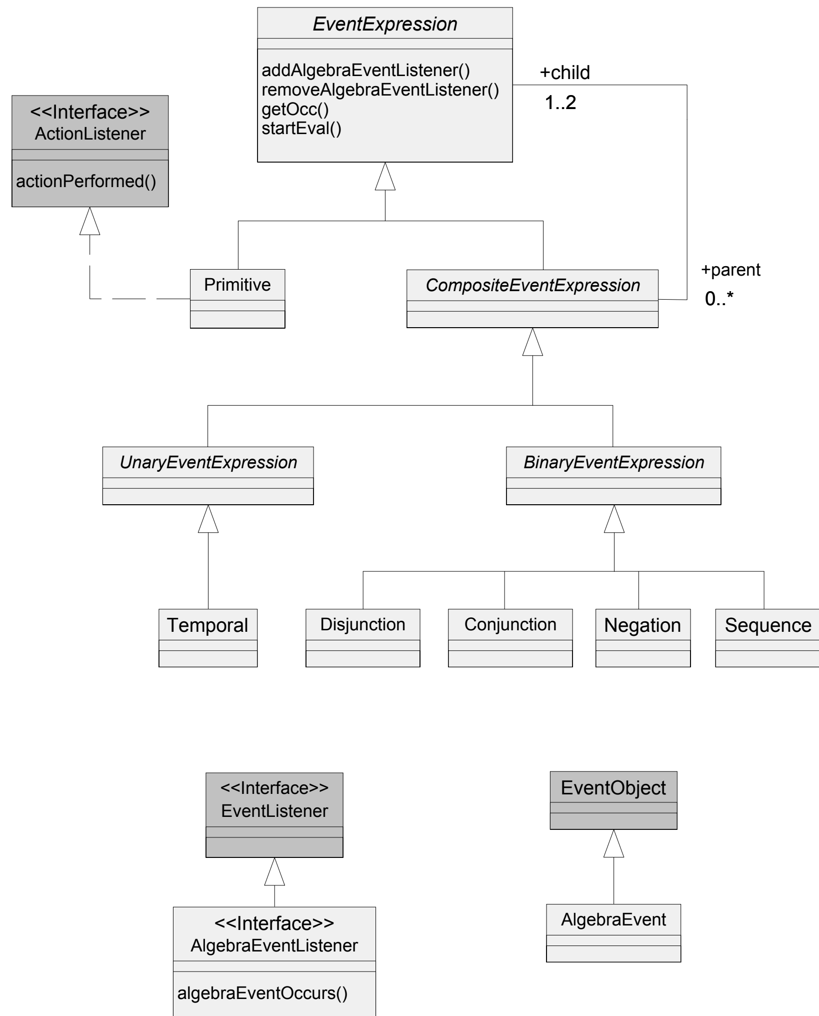


Figure 6.1: Class diagram for the event algebra package. Dark classes denote existing Java classes and interfaces.

ues when constructed. Occurrences that violate this are ignored by the primitive event object. A default minsep value of 1 is used when no information about minimum separation is available. Sequences can be created with an additional time value indicating the maximum length of the instances of the second subexpression, corresponding to the notation $A;_\tau B$ from Chapter 5. If this value is provided, the limited sequence algorithm is used, and the maximum size of the storage structure is computed from the minimum separation of primitive events.

An expression is built from instances of the classes corresponding to the respective operators, associated through the parent/child references. A concrete example of the object structure is given in Figure 6.2.

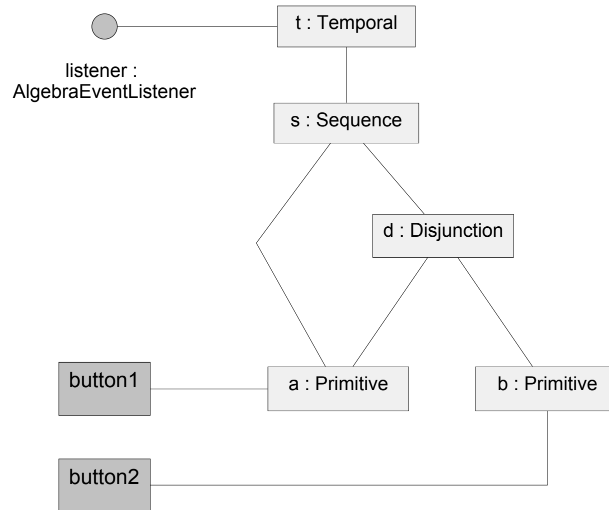


Figure 6.2: Object diagram describing the representation of the expression $(\mathsf{A};(\mathsf{A}\vee\mathsf{B}))_{5000}$ where A and B are connected to two buttons. Dark objects denote entities outside the event algebra part of the system.

The references between parent and child are navigable in both directions, since messages are sent in both directions during event processing. This causes problems if an expression is no longer of interest to the application, i.e., if there is no AlgebraEventListener connected to the expression and no variable referring to it. In such a case, the fact that

all objects are still reachable by a chain of references from some active object (the event sources) makes the Java garbage collection unable to destroy the objects and reclaim the resources allocated by the expression. A solution to this problem is provided by *weak references*. If an object is not reachable from any active object unless at least one weak reference is used, the object is said to be weakly reachable and may be finalized by the garbage collector [35]. By using weak references from child to parent, and ordinary references in the opposite direction, an expression that is of no further use can be reclaimed even if it share for example a primitive event with a currently active expression. If the application has a reference to a subexpression, this part of the object structure is not weakly reachable and thus will not be reclaimed. Thus, the application is given the responsibility of identifying active expressions by keeping explicit references to them.

We mentioned earlier that due to the lack of a central object governing an expression, the order in which an event occurrence is processed must be based on message passing between parent and child only. This causes some problems since expressions do not form trees but rather directed acyclic graphs. It is vital to the correctness of the operator implementation that if both of its arguments occur at the same time, this information is present when the operator is evaluated.

A bottom-up evaluation, as in the original detection algorithm, would ensure this but would require additional references between siblings or a separate object controlling the order in which the expression is evaluated. Both alternatives require that changes to an expression during runtime are detected, to ensure that sibling references or the control structure, respectively, are updated accordingly. Instead, we use a double traversal of the expression, illustrated by the following example.

Example 6.1. The sequence diagram in Figure 6.3 describes the communication controlling the evaluation of the expression $(\mathsf{A};(\mathsf{A} \lor \mathsf{B}))_{5000}$. The occurrence of A causes a chain of startEval messages to be sent from child to parent, indicating that the expression should be evaluated. When this message reaches a node without parents, a second traversal of the expression is initiated. Each operator recursively requests the children to compute the current instance, before doing so itself. The result is a post-order traversal of the tree, which is equivalent to the bottom-up traversal in the original detection algorithm with respect to the correctness result. During this traversal, new occurrences are also passed on to

any registered AlgebraListener.

By storing the time of the last message sent to the parent, and the last request received, duplicated computation due to common subexpressions can be avoided. In this example, the message sent from A to its second parent is ignored when it reaches the sequence. ◇

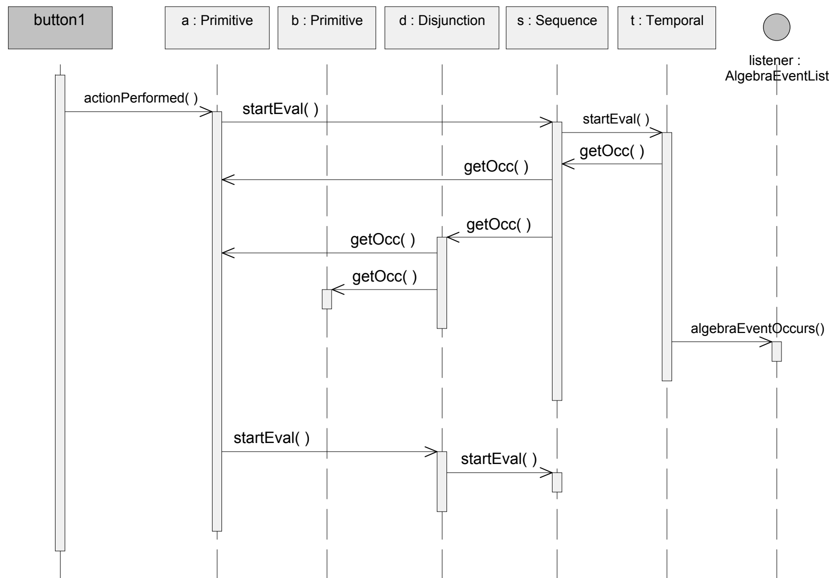


Figure 6.3: Sequence diagram depicting the communication within the expression $(\mathsf{A};(\mathsf{A}\vee\mathsf{B}))_{5000}$ when A occurs.

6.3 Using the Prototype

The algebra package is available from the following URL:

`http://www.mrtc.mdh.se/projects/HLL/Eventalgebra/index.html`

This site also contains an example Java applet where the user can enter event expressions and then generate primitive event occurrences to investigate when the given event expressions are detected.

The code snippet in Figure 6.4 exemplifies how an expression can be constructed in the prototype implementation.

```
// Declare variables
Primitive a,b;
EventExpression d,s,t;

// Create two primitive events
a = new Primitive(100);     // minsep = 0.1 sec
b = new Primitive(100);     // minsep = 0.1 sec

// Create the expression
d = new Disjunction(a,b);
s = new Sequence(a,d,0,1);  // tau=0, maxstorage=1
t = new Temporal(s,5000);

// or alternatively:
t = new Temporal(
          new Sequence(a,new Disjunction(a,b),0,1), 5000);

// Register this object as the listener of the expression
t.addAlgebraEventListener(this);

// Connect primitive events to the GUI to start detection
button1.addActionListener(a);
button2.addActionListener(b);
```

Figure 6.4: Code for creating the expression $(\mathsf{A};(\mathsf{A}\vee\mathsf{B}))_{5000}$.

Chapter 7

Related Work

A lot of work, especially formal approaches, on composite event detection has been done in the context of active databases. In addition, event detection has been addressed as a way to design large-scale distributed applications, as additional middleware functionality, and as a technique to monitor complex systems. In the area of artificial intelligence, related issues like how to represent and reason about time-dependent knowledge, have been investigated.

In general, many of the systems described here cover aspects that are not considered in this thesis, such as the specification of value constraints and actions, or how to deal with timing issues in distributed systems. In this summary we focus primarily on the event specification and detection functionality.

7.1 Active Databases

One area where event algebras are used is active databases which, unlike passive databases, react automatically to situations that arise within or outside the database. The reactions are specified by so called *event-condition-action rules* (ECA rules) stating that when a certain event occurs, and the condition is satisfied, the given action should be performed. The event part of an ECA rule can be expressed by an event algebra to allow the database to react to complex events.

The event expression language used in the object database Ode has the same expressive power as regular expressions, which allows the de-

tection mechanism to be implemented by finite state automata [20]. The definition is based on a global, totally ordered set of primitive event occurrences, implying that primitive events can not occur simultaneously. To allow event occurrences to carry values and composite events that occur only under given restrictions on the values of the constituent events, the automata mechanism is extended with data structures that store the values of events that have occurred.

In the active database SAMOS, event detection is implemented using Petri nets [18, 19]. Event occurrences are associated with a number of parameter-value pairs, and it can be specified that a complex event should occur only if the constituent event occurrences have the same value for a given parameter. SAMOS does not allow simultaneous primitive event occurrences.

Snoop [14, 13] is an event specification language for active databases. In addition to the unrestricted operator semantics, it defines four different event contexts (called parameter contexts). The operators are defined formally, but the event contexts are only described informally. The detection mechanism is based on trees corresponding to the event expressions, where primitive event occurrences are inserted at the leaves and propagate upwards in the tree as they cause more complex events to occur.

Zimmer and Unland present a formal event context framework in which the event algebras of Snoop, SAMOS, Ode and a few other systems are compared [40]. They also highlight a number of ambiguities and inconsistencies of the various approaches. In this framework, contexts are defined by two orthogonal concepts. An *instance selection* policy defines which instances should be used when more than one are considered by the unrestricted operator semantics. Then an *instance consumption* policy defines how this affects the detection of future occurrences, for example if instances are allowed to be used more than once. The selection and consumption policies are explicitly applied to the individual constituent events, making the framework more flexible than event contexts defined in terms of occurrences that initiate or terminate the composite instance.

The event algebra presented by Baily and Mikulás follows this framework [4]. 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 events whether an occurrence of the first event always implies that the other

event occurs as well.

The event detection mechanisms described above provide no assistance to a developer in terms of algebraic laws, and in general little is said about the memory and time complexity associated with the detection of complex events. The algebra presented in this thesis was originally influenced by Snoop, which is visible in the choice of operators and the use of a restriction policy. Since our ambition was to provide intuitive algebraic laws, the operator semantics differ to some extent, but the main difference lies in the way restriction is applied. In our approach, the restriction policy is designed to be applied to the expression as a whole, rather than to the individual operators, resulting in a less complex semantics for nested operators.

7.2 Event Monitoring in Real-Time Systems

A formalised schema for composite event detection, including a definition of the operations and event contexts of Snoop using this schema, has been defined by Mellin and Andler [29]. The operators have definitions parameterised on event contexts, which facilitates formal reasoning about the operators with different contexts applied, without requiring explicit definitions for each combination of operator and context. Recent work by Mellin includes Solicitor, a full event specification language based on Snoop, that has been designed within the proposed schema [28]. The language targets real-time systems in particular, and achieves predictable resource requirements by ensuring that resource bounds can be derived for events with an expiration time, for any combination of operator and context. This corresponds to our observation that any event on the form A_τ can be correctly detected with bounded resources in our algebra. However, thanks to the transformation algorithm many expressions not on this form can be detected with bounded resources as well.

Solicitor is similar to the algebra proposed in this thesis in several respects. Both are based on the interval semantics of Galton and Augusto [17] and use temporal restriction to ensure bounded resources. In addition to Solicitor being a full event specification language, an important distinction is that our algebra is designed to provide intuitive and well documented algebraic laws.

Liu et al. use Real Time Logic to define a system where composite events are expressed as timing constraints and handled by general tech-

niques for monitoring of timing constraints [27]. They present a mechanism for early detection of timing constraint violation, and show that it is possible to derive an upper bound on the length of the structures needed to correctly detect an event. In general, the time complexity of detection is in $O(n^3)$, but for a certain subset of expressions, an algorithm with linear complexity is possible [32, 33]. They also show how their technique can be applied to the area of active databases [27]. The constraints can be defined over event instances, which for example allows the definition of an event that occurs when the ith instance of A occurs after the ith instance of B. As a result, the technique is highly expressive compared to algebras over event types, such as the one presented in this thesis.

Bækgaard and Godsken [3] present a language based on TCCS [39], similar in style to the process algebra CCS [31], targeting the ECA paradigm. Triggering events are defined as processes, possibly composed from several communicating subprocesses executing concurrently, that synchronise on primitive events to detect a certain pattern.

7.3 Event Notification Services

Large, distributed systems can be designed based on an architecture of event subscribers and publishers. In such a system, rather than having subscribers register their interest in a number of primitive event types, and perform their own filtering and composite event detection, this functionality can be provided by the publisher. The subscribers register event patterns, specified for example in an event algebra. The publisher performs the event detection and notifies the individual subscribers when their pattern is detected. This reduces the communication within the system, and potentially gives a higher overall efficiency since the publisher can perform some of the event detection for multiple subscribers concurrently when the registered event patterns overlap.

Many systems of this type has been proposed. Since most of them do not target resource-conscious applications in particular, they generally do not provide resource bounds. Also, none of the studied systems provide algebraic laws for the operators. The READY event notification service by Gruber et al. contains a simple event algebra for registering composite events. [21]. Hayton et al. describes an object-oriented system with an event algebra that is implemented by nested pushdown automata [22]. The event algebra developed by Hinze and Voisard is de-

signed to suit general event notification service systems [24, 23]. Their algebra provides time restricted sequence and conjunction, and a construct to detect the ith occurrence of a given event. Following the framework presented by Zimmer and Unland [40], the algebra is parameterised with respect to policies for event instance selection and consumption.

Middleware, i.e., software located between a client and a server or between an application program and a network, often contain event detection functionality to deal with high volumes of event occurrences. In many applications a single exceptional event, for example a network failure, causes a burst of low-level failure event occurrences throughout the system. Ideally, such occurrences should be correlated and reported as a single, composite, event occurrence. Sánchez et al. present an event correlation language for middleware platforms [37]. In their approach, event expressions are translated into correlation machines, a type of nested state machines that represent the state of several concurrent detection activities, similar to Petri nets.

7.4 Knowledge Representation

The interval semantics suggested by Galton and Augusto [17] was inspired by work in the area of knowledge representation where concepts similar to those of event detection are used to represent and reason about event occurrences. In general, these methods focus on defining and relating complex events and to allow logical inference to be made from facts about event occurrences, rather than on detecting composite events as they occur. They are typically highly expressive, with good algebraic properties, but do not address efficiency of detection or bounded resource requirements.

Allen's interval algebra [1] defines thirteen possible relations between temporal intervals. An algorithm is presented 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, a temporal interval logic is defined where the truth value of predicates may vary over time [2]. Event Calculus [26, 25] is similar in style to the temporal interval logic but avoids the use of non-classical logic. Events are expressed by means of Horn clauses, augmented by negation as failure, and can be executed as Prolog programs.

Chronicle recognition [15, 16] addresses the detection of a certain type of event patterns. A chronicle is a disjunction of event sequences,

possibly with limits on the time between events in a sequence. To deal with complexity, a solution similar to event contexts is suggested where only non-overlapping occurrences of a chronicle are reported.

Chapter 8

Conclusions

The algebra proposed in this thesis is based on a straightforward, declarative semantics. We have presented a number of algebraic laws that facilitate formal reasoning and support the claim that the algebra operators behave according to intuition also under composition. A formal restriction policy was designed to define the constraints that an implementation of the algebra must satisfy.

The restriction policy, as well as the algebra semantics, was carefully designed to ensure that the algebra can be efficiently implemented. To show this, an imperative detection algorithm was presented and analysed with respect to memory requirements and execution time complexity. We also identified a class of expressions that can be detected with limited memory. An algorithm, based on the algebraic laws, was described by which many expressions can be transformed to meet the criteria for limited memory requirements while retaining their semantic meaning.

Finally, we presented a prototype implementation that combines the algebra with the event system in Java. Primitive events can be associated with event sources in a graphical user interface, e.g., buttons or text fields, and objects register their interest in events defined by expressions in the same way as they register with ordinary event sources.

The algebraic laws identify expressions that are equivalent with respect to the algebra semantics without restriction, but an implementation should correspond to the algebra semantics with restriction applied. Thus, it is essential to establish to what extent the algebraic laws can be used when reasoning about the behaviour of an implementation.

A designer can use the straightforward semantics of the algebra, and the algebraic laws, to reason about the meaning of a given expression A. The restriction policy ensures that at any point in time that A occurs according to the algebra semantics, the detection algorithm detects exactly one of the occurrences. This still holds when A is transformed into an equivalent expression A'. If $A \equiv A'$ holds, the result of detecting A' is guaranteed to be a valid restriction to $[\![A]\!]$. Thus, at any time A occurs, exactly one of these occurrences are detected by an implementation detecting A'.

8.1 Future Work

A number of future research directions have been discussed. Some of them concern extensions to the algebra, in terms of new operators and constructs, and others consider the larger perspective of how this algebra is to be used in concert with the rest of the reactive system.

8.1.1 Evaluating the Algebra

The design of an event algebra with limited resource requirements was originally motivated by the demands of real-time and embedded systems. A natural next step would be to evaluate the proposed algebra with respect to this type of systems, and in particular to identify what extensions to this basic set of operations might be required to allow typical events of this domain to be expressed.

It would also be interesting to evaluate the algebra with respect to other applications domains, for example active databases and middleware platforms.

8.1.2 Delay Operator

The temporal restriction construct can be used to define events with timeouts, but not to define the timeout event itself. For example, we can define $(A;B)_\tau$, but not the complement event of an A occurrence that is *not* followed by B within τ time units.

To allow this type of timeout events, the algebra could be extended with a delay operation (here denoted by the symbol $\triangleright$) similar to the one used by Mellin [28]. 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 timeout event in the example above could be defined as $(A \triangleright \tau) - B$.

Extending the algebra in this way requires that the formal semantics of the operator is defined in such a way that the general algebraic properties still holds. Also, new properties for this operator must be proved to ensure that it behaves according to intuition.

8.1.3 Dealing with Values

Values do not play a significant role in the current algebra. The values from primitive event occurrences are simply merged when instances of composite events are constructed. At the top level, we assume that the values are handed over from the event detection mechanism to the part of the system that reacts to the detected event.

Many existing event algebras allow event expressions that describe value dependent events. For example, an occurrence of A with a value lower than 5, or an occurrence of A followed by a B occurrence with the same value. Value dependent constructs include filtering an event stream with respect to a predicate over the values of instances, and the possibility to assign values when instances are created.

This is not yet supported by the algebra in this thesis. Of course, filters can be applied outside the algebra, either to the primitive event streams or to the result of the algebra before it is handed over to the reactive part of the system. From the point of view of the algebra, each filtered primitive is simply regarded as a separate primitive and thus does not affect the algebra much. If filters are applied to the result, however, the restriction policy might result in unexpected results. For example, imagine detecting the expression $A; A$ and then applying a filter specifying that the two A instances must have the same value. Then, three consecutive occurrences of A with values 1, 2 and 1, do not result in a detection since the restriction policy only considers two of the three $A; A$ instances to be valid.

As future work, we consider investigating how filters can be used within expressions, and to define filters with proper algebraic properties. Also, it might be useful to provide means to specify how instances of composite events are created. Keeping only some values from the simpler instances, or computing a new value from them, does not change the properties of the algebra as long as the start and end times are consistent with the current semantics, but might increase the efficiency of

an implementation.

8.1.4 Implementing the Algebra

The event algebra proposed in this thesis forms the core of an event detection mechanism, but additional work is required to turn it into a sharp implementation. A general event detection mechanism should provide an interface for defining the primitive events of the system, possibly including information about minimum separation and type definitions for values. The interface should also allow new composite events to be registered for detection, and removal of existing events.

Handling events with common subexpressions is facilitated if the transformation algorithm can create duplicates of common subexpressions when this allows a more efficient detection. For some applications it might be useful to allow existing expressions to be dynamically modified without resetting the stored information. For example, a mode change in the system might affect the time value of a temporal restriction, but events that span the mode change must still be detected. This also requires extensions to the transformation algorithm.

We intend to investigate how the event algebra can be combined with existing languages that specifically target reactive and real-time systems. The prototype implementation of the algebra is based on the J2ME Personal Profile, a subset of Java suitable for resource-constrained devices. The Real-Time Specification for Java [7] describes additions to the Java platform that allow Java programs to be used for real-time applications. We would like to examine how the event algebra can be used together with this extended Java platform.

The functional reactive language suite AFRP [38, 34], including the robot programming language FROB [36], is based on time varying behaviours and discrete events. We belive that an efficient implementation of the event algebra in AFRP would facilitate development as well as formal reasoning about AFRP programs.

Timber [12] is an object-oriented reactive language with a purely functional core, designed to target embedded systems in particular. It should be reasonably straightforward to develop a Timber implementation of the algebra from the current, object-oriented, Java implementation.

Another possible direction is to combine the algebra with the synchronous language Esterel [5, 6]. Esterel is built around a notion of

discrete time intervals during which a number of events can occur that should be reacted to. Incorporating the event algebra so that the Esterel program reacts to occurrences of composite events rather than primitive would separate pure event detection from the rest of the application logic, which would hopefully increase readability and provability of programs.

Bibliography

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

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

[3] L. Bækgaard and J. C. Godskesen. Real-time event control in active databases. *Journal of Systems and Software*, 42(3):263–271, 1998.

[4] J. Bailey and S. 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.

[5] G. Berry. *The Esterel-V5 Language Primer*. CMA and Inria, Sophia-Antipolis, France, v 5.21, release 2.0 edition, May 1999.

[6] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 2000.

[7] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[8] J. Carlson and B. Lisper. An interval-based algebra for restricted event detection. In Kim G. Larsen and Peter Niebert, editors, *First Int. Workshop on Formal Modeling and Analysis of Timed Systems*

(FORMATS 2003), volume 2791 of *Lecture Notes in Computer Science*, Marseille, France, September 2003.

[9] J. Carlson and B. Lisper. An event detection algebra for reactive systems, April 2004. Submitted.

[10] J. Carlson and B. Lisper. An event detection algebra for reactive systems. Technical Report MDH-MRTC-117/2004-1-SE, Dep. of Computer Science and Engineering, Mälardalen University, Sweden, April 2004. Available from: http://www.mrtc.mdh.se.

[11] J. Carlson and B. Lisper. An improved algebra for restricted event detection. Technical Report MDH-MRTC-159/2004-1-SE, Dep. of Computer Science and Engineering, Mälardalen University, Sweden, February 2004. Available from: http://www.mrtc.mdh.se.

[12] M. Carlsson, J. Nordlander, and D. Kieburtz. The semantic layers of Timber. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS'2003)*, volume 2895 of *Lecture Notes in Computer Science*, Beijing, China, 26–29 November 2003. Springer-Verlag.

[13] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. 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.

[14] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[15] 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.

[16] Christophe Dousson. Alarm driven supervision for telecommunication networks: II- On-line chronicle recognition. *Annals of Telecommunications*, pages 501–508, October 1996. CNET, France Telecom.

[17] A. Galton and J. C. 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.

[18] 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.

[19] 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.

[20] 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.

[21] R.E. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the READY event notification service. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, Middleware Workshop*, Austin, TX, USA, May 1999.

[22] R. Hayton, J. Bacon, J. Bates, and K. Moody. Using events to build large scale distributed applications. In *Proceedings of the ACM SIGOPS European Workshop*, 1996.

[23] A. Hinze and A. Voisard. Composite events in notification services with application to logistics support. Technical Report tr-B-02-10, Freie Universitaet Berlin, 2002.

[24] 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.

[25] R. Kowalski. Database updates in the event calculus. *The Journal of Logic Programming*, 12:121, January 1992.

[26] R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[27] 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.

[28] J. Mellin. *Resource-Predictable and Efficient Monitoring of Events.* PhD thesis, Department of Computer Science, University of Skövde, June 2004.

[29] J. Mellin and S. F. Adler. 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.

[30] Sun Microsystems. J2ME Personal Profile. http://java.sun.com/products/personalprofile/. Accessed 27 Apr 2004.

[31] R. Milner. *Communication and concurrency.* Prentice-Hall, 1989.

[32] 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.

[33] 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.

[34] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (HASKELL-02)*, pages 51–64, New York, October 3 2002. ACM Press.

[35] Monica Pawlan. Reference objects and garbage collection. http://java.sun.com/developer/technicalArticles/ALT/RefObj/, August 1998.

[36] I. Pembeci, H. Nilsson, and G. Hager. Functional reactive robotics: an exercise in principled integration of domain-specific languages. In *Proc. 4th ACM SIGPLAN Conference on Principles and Practice of*

Declarative Programming (PPDP-02), pages 168–179. ACM Press, October 6–8 2002.

[37] 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.

[38] Z. Wan and P. Hudak. Functional reactive programming from first principles. *ACM SIGPLAN Notices*, 35(5):242–252, May 2000.

[39] W. Yi. Real-time behaviour of asynchronous agents. In *Proceedings on Theories of concurrency : unification and extension*, pages 502–520. Springer-Verlag New York, Inc., 1990.

[40] 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.

Appendix A

Experimental Results

Here, we present more detailed results for the two experiments described in Section 5.2. Expressions containing N operators were created randomly, with equal probability for the five operators to occur. For temporal restrictions, random time values were generated uniformly between 0 and a given limit T.

First, we assume that no information about minimum occurrence separation is available for the primitive events. Then, the experiment is repeated with half of the primitive events having minsep values uniformly generated between 2 and 50.

Each configuration is represented by 10,000 expressions. The results are presented in diagrams where storage size, measured in number of instances, is represented on the x-axis, and the y-axis indicates how many of the 10.000 expressions that require a particular storage size. For some configurations, especially when transformation is not applied, the required storage size exceeds 2,000 instances for a few expressions. To make the significant parts more readable, only storage sizes up to 250 are included in the diagrams.

A.1 Experiment 1

For each expression the transformation algorithm is applied and the resulting expression is analysed with respect to the maximum number of instances that must be stored by the detection algorithms in order to detect the expression correctly. The diagrams in Figures A.1 and A.2

present the results.

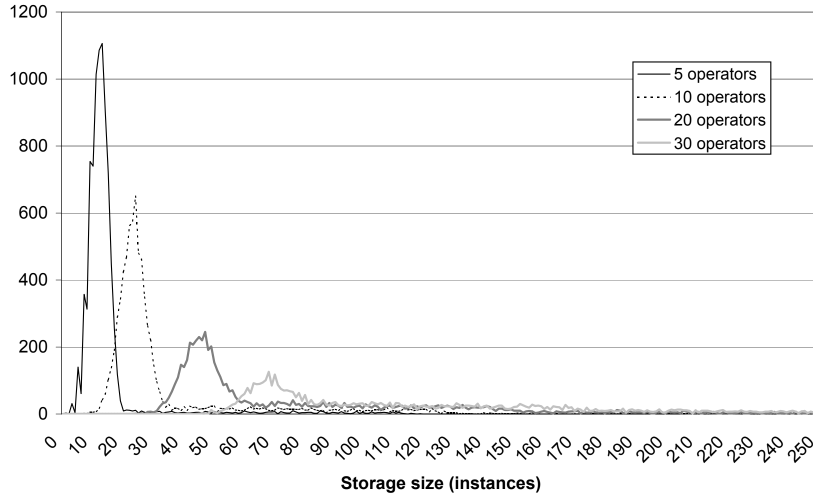
A.2 Experiment 2

The second experiment is restricted to expressions where all sequences are on the form $(A;B)_\tau$, which ensures that the detection can be performed with limited resources. Results are presented in Figures A.3 to A.6. Each figure contains two diagrams, with the storage size before and after the transformation algorithm is applied, respectively.

When transformation is applied, the number of expressions with low storage size requirements increases dramatically. The total number of expressions is the same in all diagrams, but the corresponding decrease of large storage sizes is scattered over a large part of the x-axis, and thus it is less noticeable in the diagrams. The main cause of this storage size decrease is that the temporal restriction of many sequence operators can be decreased when the transformation algorithm is applied.

The sawtooth effect in the diagrams for expressions after transformation indicates that odd storage sizes are more common than even sizes, especially for small expressions. The same effect is visible also in the first experiment, but it is much more prominent for the class of expressions used in this experiment.

For a brief explanation to this, we note first that a primitive event requires an odd number of instances (one) to be stored. If we consider the ways in which expressions of odd storage size can be combined into a composite expression, we notice that the storage size of the resulting expression is more often odd than even. One contributing factor is that most temporal restrictions are removed by the transformation algorithm. The imbalance is even larger for the class of expressions used in this experiment. For general expressions, sequences often result in an even storage size when the subexpressions have odd storage sizes, but here they always occur in a temporal restriction that makes the storage size odd again.



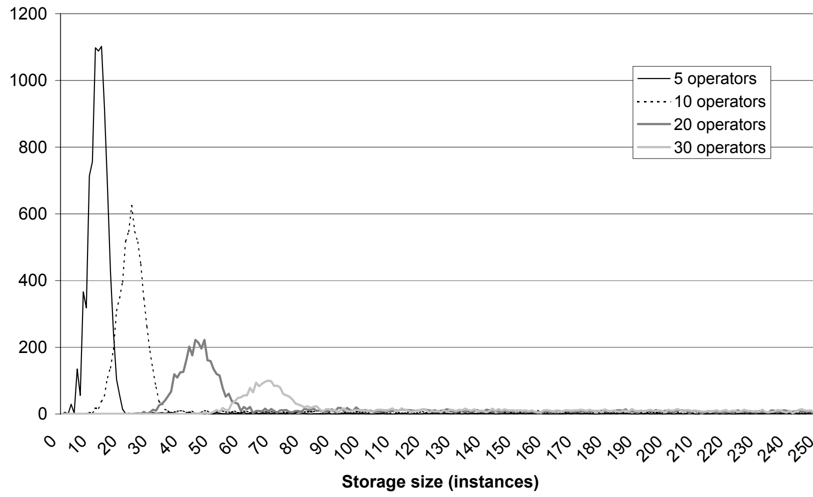
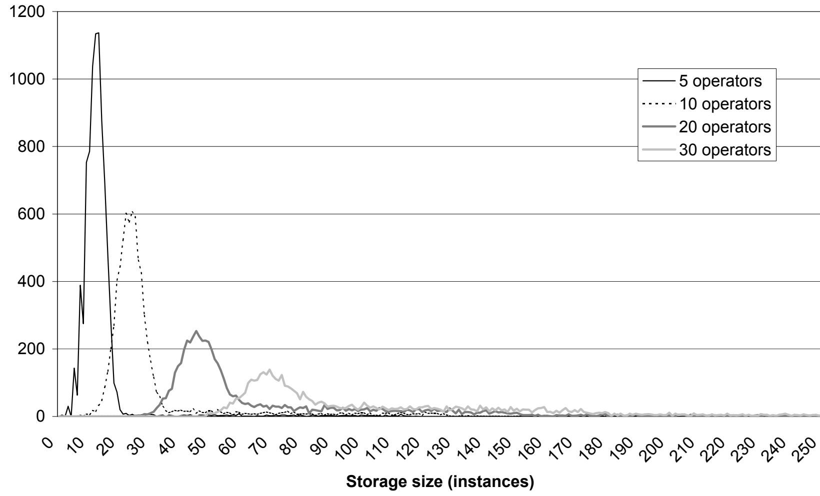


Figure A.1: Experiment 1a: Storage requirement when and all primitive events have a minsep value of 1, for $T = 100$ (top) and $T = 300$ (bottom).



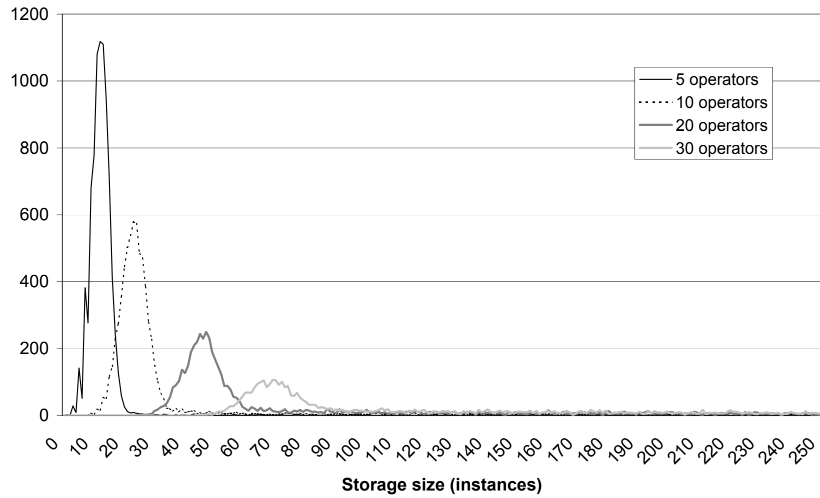


Figure A.2: Experiment 1b: Storage requirement when half of the primitive events have random minsep values between 2 and 50, for $T = 100$ (top) and $T = 300$ (bottom).

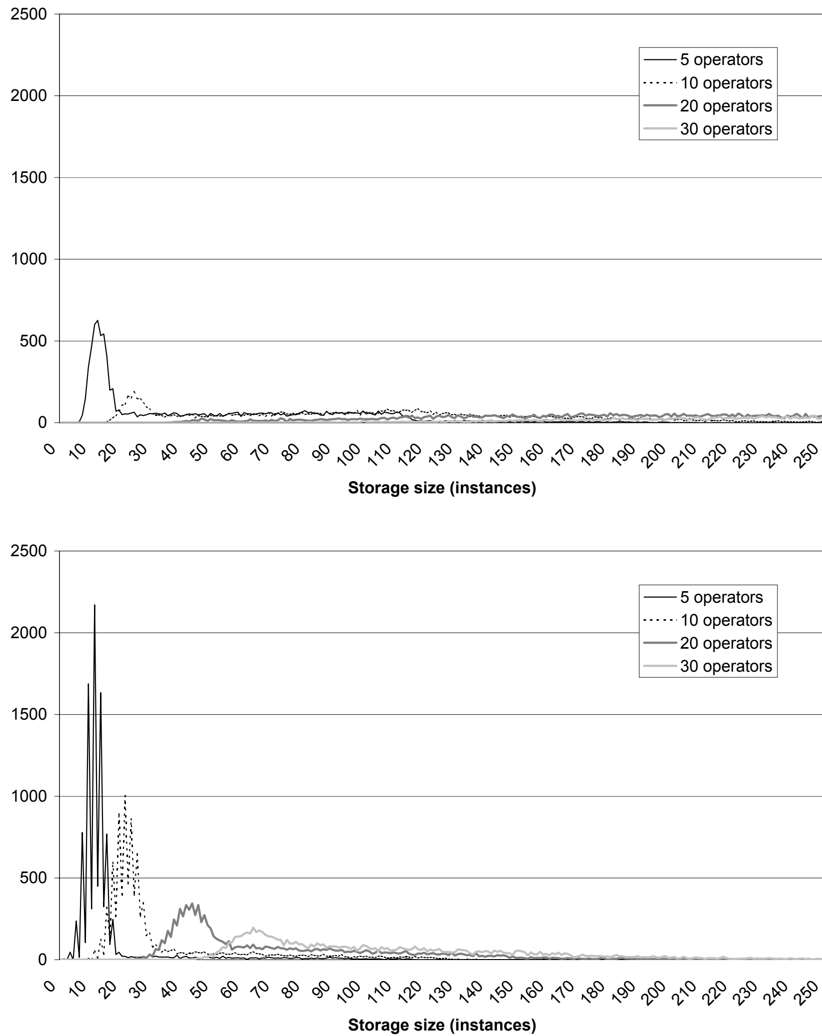


Figure A.3: Experiment 2a: Storage requirement before (top) and after (bottom) transformation, when $T = 100$ and all primitive events have a minsep value of 1.

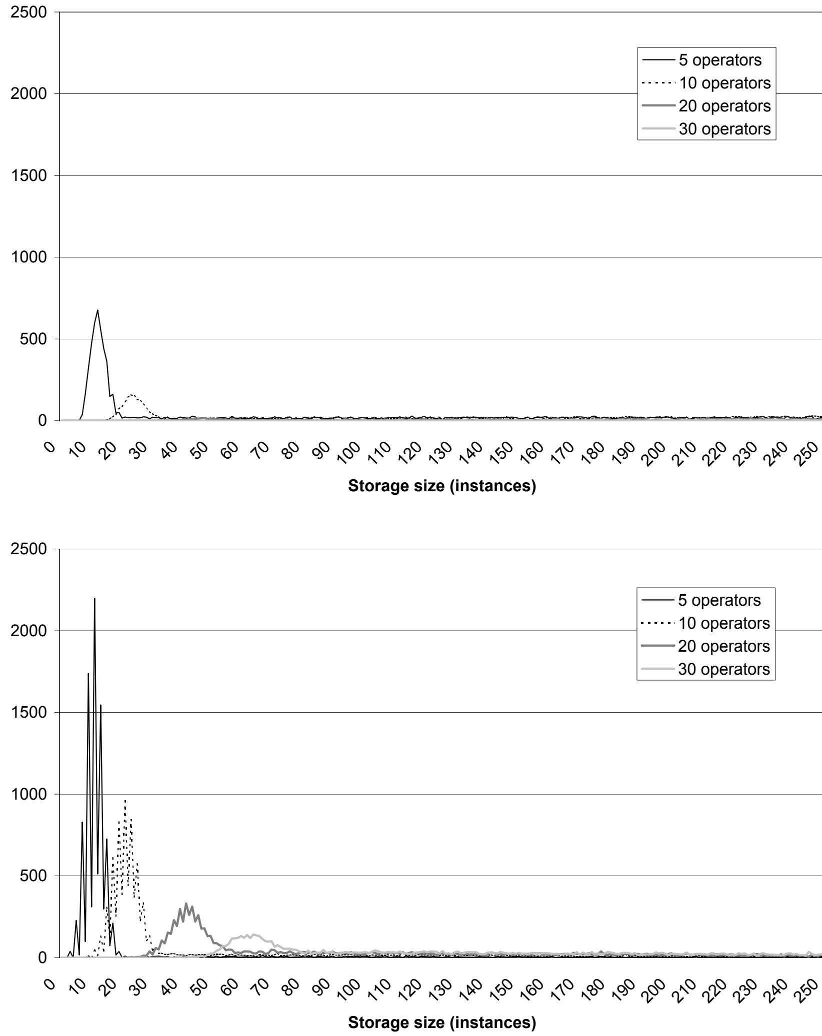


Figure A.4: Experiment 2a: Storage requirement before (top) and after (bottom) transformation, when $T = 300$ and all primitive events have a minsep value of 1.

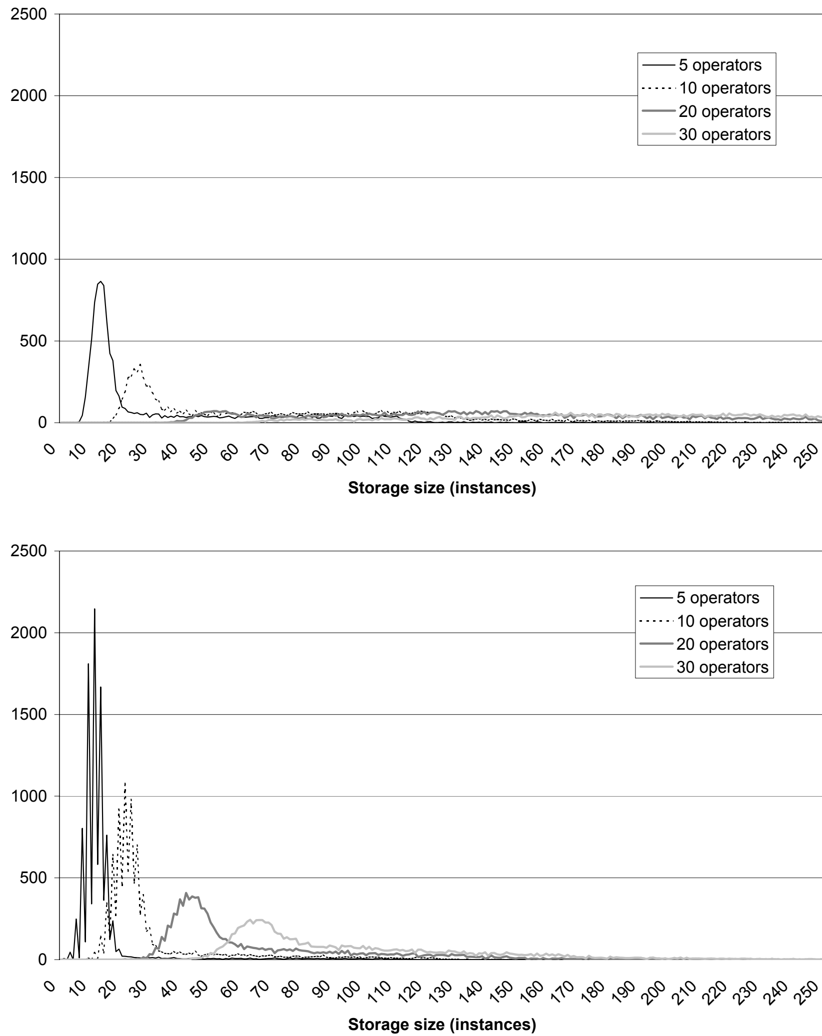


Figure A.5: Experiment 2b: Storage requirement before (top) and after (bottom) transformation, when $T = 100$ and half of the primitive events have random minsep values between 2 and 50.

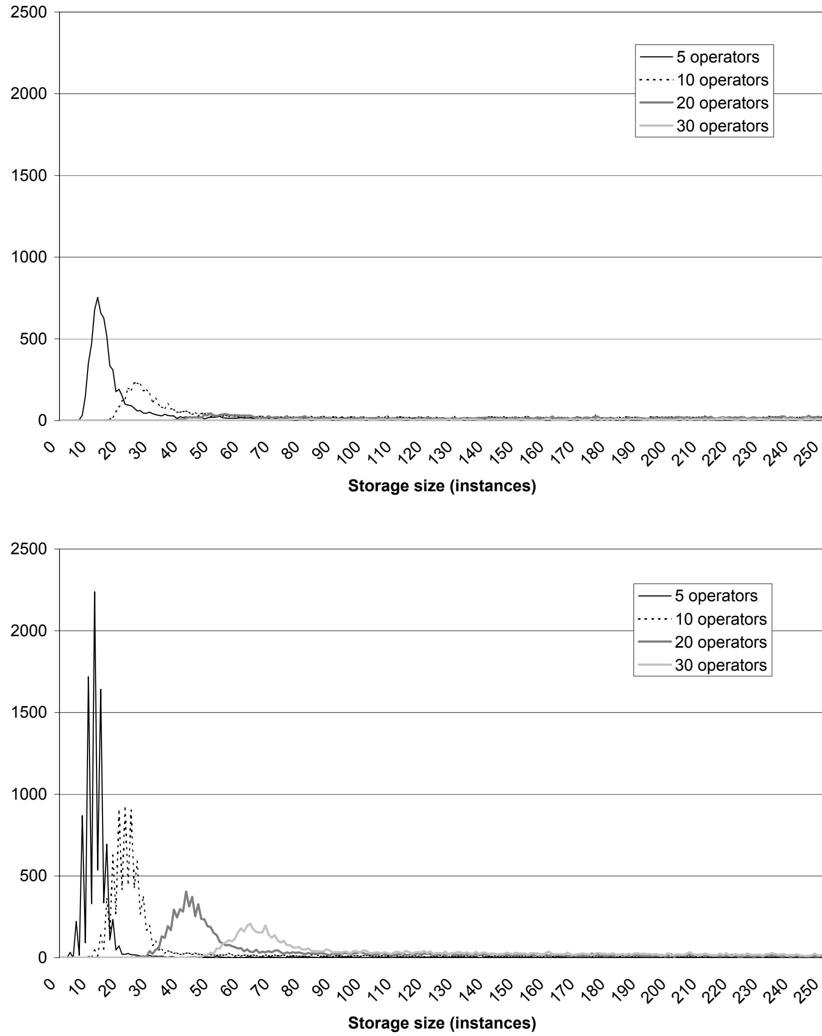


Figure A.6: Storage requirement before (top) and after (bottom) transformation, when $T = 300$ and half of the primitive events have random minsep values between 2 and 50.